AD-A230 284

CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: **Final Report - Real-Time Performance Benchmarks for Ada**

CIN: **C02 092LY 0001 00**

24 MARCH 1989

Real-time Performance Benchmarks

For Ada

Contract Number: DAAA21-85-C-0238

Prepared For:

U.S. Army, CECOM
Advanced Software Technology
AMSEL-RD-SE-AST-SS-R
Ft. Monmouth, NJ 07703-5000

Prepared By:

Arvind Goel
TAMSCO
145 Wyckoff Road
Eatontown, NJ 07724

October, 1988.

CONTENTS

LIST OF TABLES

Chapter 1: Introduction

This report develops a benchmarking capability for measuring the performance of Ada compilers meant for real-time embedded systems. Cross-compiler users face many tough trade-offs and additional project costs due to time lost developing compiler work-arounds. The embedded programmer community is faced with trying to use incomplete and somewhat unstable systems while trying to deliver efficient and ultra-reliable code. For any given real-time embedded application, the best performance predicator is the performance of the application software itself. Software designers and programmers can be satisfied that an embedded application meets its performance requirements ONLY after testing the final product. But due to the rising costs of software development, users may want to use benchmarks to determine the performance and suitability of a particular compilation system for their real-time applications. The major focus in this report is on developing benchmarks to measure runtime performance of Ada features important for real-time embedded systems.

A motivating factor in the development of Ada as the Department of Defense standard language was the high cost of embedded system software development. The principal goal of Ada is to provide a language supporting modern software engineering principles to design and develop real-time embedded systems software. Ada is said to be complete in the sense that it comprises a) the necessary features for the manipulation of low-level entities like bits, characters, and addresses, b) features for high-level abstraction, and c) features for parallelism. These features of Ada should enable the development of a product to be

- more cost-effective,
- more maintainable over its lifetime,
- and more portable between different systems.

Unfortunately, most of the current Ada implementations do not allow the development of embedded systems software reliably and without sacrificing productivity and quality. A prime example is avionics software applications which demand very fast processing, swift calculation of floating-point math, and compact object code. Current Ada compiler implementations are unable to support these demands due to several reasons, some of which are:

- they are written by software engineers with experience in large-system design,
- lack of operating system knowledge and real-time issues,
- more concern with passing the Ada Compiler Validation Capability Test suite,
- implementation and size of the Ada runtime system which differs widely from one compiler to another. An example of inefficient runtime system implementation is the bringing of complete runtime systems from the host computer to the target without removing such items as disk controllers, screen drivers and other I/O interfaces that may be unnecessary for most embedded systems.

The Ada Language Reference Manual (LRM) has a lot of implementation dependent features that are of concern to real-time programmers. The large variance in implementation options for a feature effect application program behavior and efficiency (the list of the implementation dependent features is compiled in a document "Catalog of Ada Runtime Implementation Dependencies" published by the Ada Runtime Environment Working Group (ARTEWG) [4]). The implementation dependencies are a clear signal that simply adopting the language as defined in the LRM is not enough for real-time embedded systems. The performance and implementation approach of various Ada language features and the runtime system has to be benchmarked to assess an Ada compiler's suitability for a real-time embedded application. Some of the problems that arise in benchmarking Ada compiler systems are:

- Ada benchmarking is much more complex from other languages because of the powerful and sophisticated runtime system that supports Ada features such as memory management, process scheduling and control, tasking, etc. and whose implementation varies from one compiler system to another.
- Significant complications also arise in Ada in defining details of what is to be measured and in achieving those measurements.
- Selecting a benchmarking technique that ensures the accuracy and relevance of the benchmark results is non-trivial.

At this point, it will be useful to briefly describe the definition and requirements of real-time systems in order to get a better understanding of the effort involved in developing benchmarks for embedded systems.

1.1 Real-time Embedded Systems

A real-time embedded system constantly monitors, analyzes and responds to external real world events in a time critical fashion. Embedded systems usually consist of specialized hardware running dedicated application software at the heart of a real-time control or data-processing system and the application software uses the entire assets of the computational system to solve a problem. Also, embedded systems usually have no operating system support and often consist of the minimum configuration of CPU, memory and peripheral interfaces. Examples of embedded systems are aircraft avionics systems, ship weapons control systems, and engine control systems.

1.2 Requirements Of Real-time Systems

Real-time embedded systems have severe timing and memory constraints. In an embedded application, the time taken to perform system functions such as process initiation, process termination, and context switching is crucial. System start-up time is also important as is the time taken to change operating modes, to reconfigure the system after a partial failure, or to restart the system after a total failure. For a programming language to be used effectively to program real-time embedded systems, it should be able to support the following characteristics [13]:

- Periodic tasks: Many physical processes controlled by embedded computers require tasks to run periodically on a scheduled time basis. Given the need for periodic tasks, a convenient and meaningful method of expressing timing behavior should be provided.
- Sporadic Tasks: are synchronous events (that arrive randomly) and must have guaranteed response times. A sporadic task may arrive randomly. However, there is a worst case (maximum) arrival rate and execution time that can be used to guarantee the response to critical events.
- Fault Tolerance: Software faults are design faults and are unanticipated unlike hardware faults. The software should be able to recover gracefully from unanticipated faults.
- Distributed Systems: Many embedded computing systems depend on the coordinated activity of a set of computing devices rather than a single computer. Programming language semantics for managing the distribution of a program and of time across a network of processors are necessary.
- Time Abstraction: Timing is the most critical aspect of a real-time system. A programming language should have abstractions for deadlines, time-out exceptions, delay, and timed-accept constructs. An embedded system must provide specified amounts of computation within required time intervals.
- Reconfiguration: is event-driven reconfiguration where a separate activity is started at the beginning due to mission requirement changes. Such reconfiguration are generally planned for at design time.
- Resource Utilization: A programmer should be able to predict and ideally explicitly control a program's utilization of resources including memory and execution time. Where there exist alternate ways of implementing an Ada construct, a way should be provided for the programmer to obtain the translation that best meets the needs of the application.
- Concurrency: Real-time embedded applications are inherently parallel in nature. The extensions of a model of concurrency into the programming language therefore provides abstractions for use in the engineering of the software.

- Reliability: An embedded system may have to operate nonstop for an extended period of time. Predictable exception handling for expected errors is required and tolerance for unexpected errors is also desirable.
- Traceability: For debugging purposes, extensive knowledge of how each programming language construct is transformed into machine code is required.

Languages for programming real-time embedded computer systems, in general, require the ability to represent concurrent control of separate system components, the structures needed to build extremely reliable systems, the ability to specify in real-time when actions are to be performed, and the ability to interact with special purpose hardware.

1.3 Ada and Real-time Systems

In traditional real-time systems, an executive running on the system was responsible for making sure that the various timing and memory constraints were satisfied by different parts of an application program. This executive was typically coded in assembly language and was designed and tailored to meet the timing and memory requirements of the application at hand. The executive used in a real-time application could not be used in another application, and programmers had to typically write an executive for each real-time application. The programmers also had complete control over all actions and resource allocation.

In Ada the executive is part of the language as the Ada runtime system. Real-time programmers generally have no control on the design and implementation of the Ada runtime system except for the fact that the runtime system satisfy the requirements listed in the LRM. For example, the LRM states that a task will be suspended for at least the time specified in a delay statement, but does not state when after expiration of that delay is the task eligible for execution. This could be very critical in real-time embedded systems where tasks may have to be executed at a fixed time in order to satisfy the timing constraints of the system. The ability of Ada to satisfy timing and memory constraints of embedded real-time systems has to be demonstrated if Ada is to be employed successfully in embedded systems.

1.3.1 Ada Runtime System

The Ada runtime system implements the code to support Ada semantics. It appears

as object code at execution time and provides many of the support functions previously designed and written by embedded applications designers. In embedded systems, the runtime system is responsible for dynamic memory management, interrupt management, time management, exception management, rendezvous management, task activations and terminations, I/O management and scheduling the various tasks in the application program. For real-time systems, object code and the supporting runtime system should be fast, compact and configurable. For some embedded applications, it may be necessary to tailor and/or configure the runtime system so that the user does not have to pay time and space performance penalties for features of the language that are not used.

Many of the reasons that embedded systems designers may be dissatisfied with Ada are related in some way to the current runtime system implementations as well as the lack of certain features in the language:

- runtime system is too big and slow. It has a large number of implementation dependent characteristics which present a) portability problems for embedded applications and b) performance inconsistencies among validated compilers which makes it difficult to select the appropriate Ada compiler for embedded applications.
- high execution time overhead,
- lack of timing predictability,
- lack of control over resource management decisions that affect system timing and reliability,
- no clear abstraction of time. The delay statement gives only a minimum delay, but it is maximum delays that are of interest.
- no abstraction of configuration, e.g. for distributed systems,
- priority mechanism can allow priority inversion, e.g. a low-priority task running during rendezvous with a high-priority task can lock out a medium-priority task.

Some of the problems identified can be circumvented by the implementation of the proposals presented by ARTEWG [6], but others (for example, high execution time overhead and runtime implementation dependencies) are compiler dependent and need to be benchmarked.

1.4 Scope of this Report

To determine the suitability of Ada compiler systems for embedded applications, a benchmarking effort has been undertaken by the Center for Software Engineering, Ft. Monmouth, NJ. As part of this effort, existing benchmarks have been researched to

determine their suitability for determining the performance of real-time embedded systems. Using the University of Michigan [1] benchmarks as the starting base, existing benchmarks have been modified and new benchmarks developed to measure the performance of Ada features important for real-time embedded systems. In another effort undertaken by the Center for Software Engineering, a matrix which maps real-time Ada features to benchmarks was developed [12]. In the current effort, this matrix has been used to develop additional benchmarks for those Ada characteristics which have no benchmarks available. Some Ada features cannot be benchmarked reliably and the reasons have been explained in the appropriate sections. The scope of this benchmarking effort is to determine

- the runtime performance of Ada code on a bare target system,
- the runtime system implementations of various features of a particular Ada compiler system,
- and the performance of commonly used Ada real-time paradigms that may be programmed using macro constructs (a macro construct is defined as a set of Ada statements that perform a well defined process e.g. semaphores [12]).

For this benchmarking effort, it is assumed that the runtime system is written for a bare target and is fully responsible for runtime performance and efficiency. The target is a uniprocessor and the problems of multi-processing and multiprogramming systems are left to be dealt with elsewhere.

1.4.1 Testbed Hardware and Software

The hardware used for benchmarking was Sun 3/60 CPU running Sun Unix 4.2 Release 3.5, linked to a single 12.5 Mhz Motorola 68020 single board computer enclosed in a multibus chasis. The setup can be summarized as follows:

Host: **Sun 3/60, running Sun Unix 4.2 Release 3.5**

Compiler: **Verdix Ada Development System targeted to Motorola MC68020 targets, release 5.41**

Target: GPC68020 (based on Motorola MC68020 microprocessor) multibus-compatible computer board having 12.5 Mhz MC68020 microprocessor, a MC68881 floating point co-processor, and 2 megabyte of RAM.

The GPC68020 has two serial lines with one RS232 line and and one line which may be configured as either RS232 or RS422. The second serial port was connected to the

SUN 3/60 serial port and used for downloading object code to the GPC68020 computer. The benchmarks were compiled on the SUN 3/60 workstation and then downloaded to the bare target and executed via tools available from the compiler vendor. The Verdix compiler cross-compilation system contained tools for compiling, linking, downloading, and executing target programs. There was also a cross-debugger that enabled debugging of programs running on the target. Command files to compile, link, download and execute the benchmarks were written. The benchmarks were compiled without the optimize option and the timings listed are for un-optimized runs. The Verdix compiler has 9 levels of optimizations available and it is impossible to compile with all of the optimization levels. The link phase included commands to define the memory layout, e.g. program placement, stack and heap sizes, etc.

1.4.2 Report Layout

This report is divided into 6 chapters (including the current one). The second chapter deals with the whole issue of Ada benchmarking. Methods of developing benchmarks as well as timing issues are discussed.

Chapter 3 deals with microscopic benchmarks. Ada features important for real-time systems are highlighted and benchmarks have been developed for those features.

Chapter 4 deals with runtime implementation dependencies benchmarks.

Chapter 5 discusses real-time paradigms that can be programmed in Ada.

Chapter 6 concludes with some thoughts about lessons learned and more work that needs to be performed in the area of benchmarking.

Chapter 2: Ada Benchmarking

In the last year or so, Ada compiler technology has reached a state to justify their use for time critical applications. Vendors are introducing Ada compiler systems which generate code for bare targets such as the Intel iAPX88 family, Motorola 68000 family, and the MIL-STD-1750A microprocessors. As discussed in Chapter 1, cross-compiler users face many tough trade-offs and additional project costs due to time lost developing compiler work-arounds. Benchmarks have to be relied upon to select a Ada compiler for developing embedded systems software. **The use of inappropriate benchmarks (that may be designed incorrectly and hence may provide false information) to select an Ada compiler system can be downright crippling for an embedded application.** Also, factors such as complexity of Ada runtime performance, interaction of language features, as well as differing requirements of different real-time systems complicate the task of developing benchmarks that measure real-time performance for all applications.

The use made of performance measurements depends on one's purpose in conducting benchmarks. The purpose may be to compare implementations for general maturity or to determine which is most suitable for a particular application. In the first case, evaluation of benchmarks will be implicitly based on assumptions about which performance characteristics are indicative of maturity. In the second case, the performance characteristics deemed most important will depend on the requirements of the application. An implementation with a low score for a particular performance characteristic may still be most appropriate for a given application. Ada benchmarking can be approached in 4 ways:

- design benchmarks to measure execution speed of individual features of the language,
- design benchmarks that determine implementation dependent attributes
- design benchmarks that measure the performance of commonly used real-time Ada paradigms (that may be programmed using macro constructs).
- design composite benchmarks which include representative code from real-time applications.

Before designing the benchmarks, it is important to define what is to be measured and which approach is to be used to measure that feature. For example, before designing microscopic benchmarks those features of Ada have been identified that are important for real-time embedded applications and can be measured using the first approach. For each of the microscopic features that have been defined, existing benchmarks have been analyzed to determine if benchmarks are available that measure that feature accurately and correctly. If the existing benchmarks have been found to be deficient they have either been modified or new benchmarks have been developed. Also, a very important step in any benchmarking effort is the interpretation of the results produced by running the benchmarks. To help interpret

the results of the benchmarks, each benchmark is followed by a detailed discussion.

Performance characteristics considered important for a real-time embedded application depend on the requirements of the application. For a class of real-time embedded applications, it is possible to define the distinguishing characteristics of such a system. A matrix can then be developed that maps these characteristics into Ada features and then maps the Ada features into benchmarks [12]. The ultimate goal of any benchmarking effort would be to have sets of benchmarks, where each set evaluates an Ada compiler for a particular class of real-time applications. Each set should include microscopic as well as composite benchmarks that are representative code for that class of applications. Specific benchmarks that measure those Ada features can then be extracted from the general set to form a set that evaluates a compiler system for that particular class of real-time systems.

2.1 Benchmarking Approach

Benchmarks can be distinguished not only by the performance characteristic being measured but also by the approach being used to measure that characteristic. As discussed in the previous section, four distinct approaches are required to design benchmarks to determine performance of Ada compiler systems. These approaches are considered in the following sections.

2.1.1 Measure Performance Of Individual Features

This approach measures the execution speed of individual features of the language and runtime system by isolating the feature to be measured to the finest extent possible. Such benchmarks are useful in understanding the efficiency of a specific feature of an Ada implementation. For example, a benchmark that measures the time for a simple rendezvous can be run on two Ada compiler systems. Based on the results, an application can choose one compiler system over the other. The advantage of such an approach is performance evaluation without bias towards any application. These tests are useful for bottleneck analysis in which a score for a given test must exceed a stated threshold if an Ada implementation is to be considered suitable for an application.

The problems with such an approach include determining and isolating the features of the language and runtime system that are important for real-time embedded system applications. Due to the complex nature of Ada, it is very difficult to determine which features need to be benchmarked. Once the feature has been decided, isolating the

feature is another problem. This is due to the lack of precision in the translation of source code to object code making the isolation of features difficult to achieve in practice and more difficult to assume consistency from one implementation to another. Also, this approach requires a significant number of tests and the numbers produced have to be statistically evaluated to determine general performance.

The design of benchmarks intended to isolate features is often complicated by unexpected interactions. Consider the assessment of task activation and heap allocation time. Often measured separately, their interaction can be significant and complex, with significant variation between implementations. Storage allocation rules are particularly vague within the language, so that a task activation could involve a stack allocation, heap allocation or specific allocation from a pre-allocated storage collection. Task allocation time will depend upon the implementation scheme and possibly on the history of previous allocations at the time of activation. Feature interactions can be static or dynamic. Examples of static interactions include:

- Code generated for storage allocation will require inter-task lockout if the collection is shared by two or more tasks.
- Code generated for a FOR loop may depend on whether or not the loop is nested.

Examples of dynamic interactions include:

- Time for rendezvous can degrade with the number of eligible tasks due to the search and sorting involved with prioritized dispatching.
- Time for dynamic allocation can depend on the state of storage management following previous allocations due to the need to recover storage and efficiently manage the available space.

The complexity of such interactions makes the task of isolating features in benchmark construction more difficult. Benchmarks have been designed that take these complex interactions into account and determine the effect on these interactions on the performance of a feature.

2.1.2 Determining Runtime System Implementation

These benchmarks are concerned primarily with determining the implementation characteristics of an Ada Runtime System. The scheduling algorithm, storage allocation/deallocation algorithm, priority of rendezvous between two tasks without explicit priorities are some of the many implementation dependent characteristics that need to be known to determine if a compiler system is suitable for a particular real-time embedded application. Some implementation dependencies cannot be benchmarked and that information has to be obtained from the compiler vendor as

well as the documentation supplied by the vendor. A major effort in such benchmarks involves interpreting the results obtained by running the benchmarks and drawing the correct conclusions. A detailed description has been provided to help interpret the results. The ARTEWG document "Catalog of Ada Runtime Implementation Dependencies" [4] lists those Ada features that are implementation dependent. This document has been consulted extensively in determining which implementation dependencies need to be benchmarked for real-time embedded systems.

2.1.3 Real-time Paradigms

This approach also involves programming algorithms found in embedded systems. For example, a situation in real-time systems may be a producer that monitors a sensor and produces output asynchronously and sends it to a consumer. The producer task cannot wait for a rendezvous with the consumer (who might be doing something else) as the producer task might miss a sensor reading. To program this paradigm in Ada requires three tasks: a producer task, a buffer task that receives input from the producer task and sends the input to the third task:consumer task.

Macro constructs are defined as a set of Ada statements that perform a well defined process e.g. semaphores, mailbox construct etc. For real-time embedded systems, real-time paradigms can be identified and programmed in Ada using macro constructs. These benchmarks can be run on Ada compiler implementations and statistics gathered on their performance.

2.1.4 Composite Benchmarks

Rather than measuring individual features, this approach looks as much at the interaction between features as to the performance of the features themselves. Good examples of this approach involve the use of typical code segments from a given application collected into a program whose overall performance is measured (like the Ada Avionics Test Program Package developed by SofTech Inc.). The benchmarking technique used to measure composite benchmarks is end-to-end where the measured code is the entire program.

The advantage of this approach is that for a given application domain, running this benchmark on different compiler implementations enables a straightforward selection. Also there are no complex combinations of feature-by-feature evaluations to consider, and no surprises stemming from an unenlightened evaluation.

The difficulty with composite assessments is their narrow scope of usefulness and their bias towards the domain of applications from which the benchmarking code was selected. Also, no single composite benchmark can capture all the information that characterizes even a subset of the real-time applications domain.

There is room for compromise here. A benchmarking suite should consider microscopic as well as composite benchmarks as well as some intermediate assessments. Such intermediate assessments might look at typical real-time programming paradigms.

The main thrust of the current task has been to benchmark micro constructs, real-time paradigms, and to determine runtime system implementation. The area of composite benchmarks has not been addressed and will be addressed in a follow-on effort.

2.2 Benchmarking Techniques

This section discusses the overall program form and measurement techniques used in the implementation of benchmarks. Benchmarking techniques are tied very closely to the benchmarking approach being used. The following benchmarking techniques are used very frequently in the design of benchmarks:

1. **Dual loop** method is used in the design of microscopic benchmarks. Two loops are constructed, one with and one without the code to be measured. The loops are iterated X number of times and then the difference between the start and stop times for each loop is taken. Finally the difference between these two differences divided by the number of iterations gives the execution time for a feature.
2. **Single program, End-to-End:** In this technique the measured code is the entire program. This is particularly appropriate for composite benchmarks where feature isolation is not of concern.
3. **Time Stamping and Object Labeling Within a Program:** As a technique all that is required here is the collection of clock times at particular points in the program's execution, or the collection of storage information or storage addresses associated with particular objects within the program during its execution. This information can then be analyzed later to determine processing time or storage use. While it is a simple approach to measuring performance characteristics, this technique requires that a suitable means can be found for identifying the elements to be measured and that the measurement can be done accurately enough.

4. **Establishment of Timing/Storage-Use Patterns:** Closely related to the previous technique, this refers to the analysis of time stamping or object labeling within a program. In this case the interest is not in the single measured result, but rather in the pattern or the significance of differences identified as indicators of implementation approach. For example, a history of task execution times could be analyzed to deduce the task scheduling algorithm.

In addition to this classification based on the basis of program design, a broad distinction needs to be made between the use of internal instrumentation and external instrumentation.

2.3 Internal and External Instrumentation

Internal instrumentation refers to the use of Ada features to provide the measurement desired, thus yielding a truly portable benchmark. External instrumentation requires various additional measurement capabilities (e.g. various electronic instruments, operating system commands, implementation specific runtime services) along with the Ada program.

The basic measurements of interest in computer systems are time and space utilization. Timing can be measured a) by either the standard Ada package CALENDAR with its clock features, or through a custom developed clock package which draw's on the processor's real-time or other clock capabilities and b) external instrumentation employed in embedded systems debugging and test beds. The problems with accessing the internal CLOCK are discussed later on in this report, but some of them are:

- excessive time required for the clock function
- inconsistent time required for the clock function
- low resolution of the clock measurements.

Space utilization is a bit more difficult to measure, partly because it is difficult to specify what the essential data items are, but also because there is often not a good way to get these measurements. The attributes of the language ('SIZE and 'ADDRESS) can be used to provide one measurement service. An alternative approach to measuring space utilization is a form of stress testing, in which processing with associated allocations is repeated until the exception STORAGE_ERROR is raised. This could be used to determine the extent of storage available in different situations. The stress test could be repeated with different data objects and program units to provide a comparison of different usage patterns.

External instrumentation may offer greater accuracies or detailed measurements, but

it is necessarily unique to the one implementation and expensive to customize for each system to be measured. Benchmarks relying on internal instrumentation on the other hand are more portable and can be automated easily.

There are some measurements that require external instrumentation. If the dual loop technique is not used to design a benchmark (e.g. time stamping is used) then a high-resolution external clock may have to be used to provide greater accuracy. Also, times attached to an external event require external instrumentation. Interrupt latency time is defined as the time from interrupt occurrence to interrupt handler execution. There is no difficulty in time stamping within the program the start of interrupt handler execution, but some form of external instrumentation is necessary to accurately schedule or capture the time of interrupt occurrence.

In this report, internal instrumentation has been used in the design of benchmarks. This approach has been followed to make the benchmarks more portable and also to enable comparison of the benchmark results more meaningful. Some benchmarks need external instrumentation in order to get any meaningful results and these benchmarks have been identified. Such benchmarks have to be tailored to the particular hardware configuration and compiler system being used to perform the measurement.

Chapter 3: Microscopic Benchmarks

Microscopic benchmarks are designed to measure the performance of individual features of the Ada programming language. This chapter discusses two things in detail: a) the methodology of designing such benchmarks, and b) the actual benchmarks that have been developed for each Ada feature. Benchmarks have been designed for all the major Ada language features that are important for real-time embedded systems. Since optimizing compilers generate different code for the same feature depending on the context in which the feature occurs, it has been attempted to benchmark a particular feature under different scenarios. The results will demonstrate the range of performance associated with a language feature.

Benchmarks that give false results about system timing and sizing for a real-time embedded system application can have disastrous results. Hence, the importance of designing benchmarks that:

- Isolate the feature that the benchmark is designed to measure
- Thwart compiler optimizations
- Provide sufficient accuracy
- Have repeatable results

cannot be emphasized more. In the following sections, we will discuss the techniques used to design benchmarks that have the characteristics listed above. To determine these techniques, existing benchmarks were researched. The benchmarks analyzed included:

- The University of Michigan Benchmarks
- and Performance Issues Working Group [PIWG87] test suite.

After a detailed analysis of these suites, it was determined that the methodology developed by the University of Michigan is best suited for benchmarking specific Ada language and runtime features that are important for real-time embedded systems. This suite addresses the issues that are of concern when designing benchmarks some of which are : isolation of features, accuracy, and thwarting compiler optimizations. In a separate paper [11], the PIWG benchmarks and the measurement techniques adopted by the PIWG benchmarks have also been discussed. **In the present effort, the benchmarks are being designed for a dedicated embedded processor with real-time clock service.** Generally, there is no virtual memory paging or system daemons in such an environment and hence there is no operating system interference. In a multi-user, virtual memory system, other processes and the operating system interference can easily distort the measurements.

3.1 Benchmark Timing

This section discusses the strategy used for time measurements, the constraints placed on test problems to permit them to be measured, the sources of measurement errors, the steps taken to minimize errors and the error bounds.

For benchmarks that measure time values using the system function CLOCK, the ideal design would be to determine the specific feature that needs to be measured and perform that feature sandwiched between calls to the system CLOCK. The difference in time is the execution time for that feature. **For this measurement to be accurate, the resolution of the CLOCK should be considerably less than the time required by the operation to be measured.** Generally, the system clock that is available to a benchmark designer may be accurate to a tenth of a second and that is inadequate to measure events in the millisecond and microsecond ranges. Measurement of time is not an easy task and it leads to inaccuracies and misleading data. Some of the problems that have to be overcome when accessing the internal CLOCK function include:

1. **Clock Precision:** In designing portable benchmarks, one can only assume the presence of the function CLOCK in the package CALENDAR. If the precision of the CLOCK function is not very high it can cause errors in the timing measurements. The precision of the CLOCK function is determined by SYSTEM.TICK which is the basic clock period in seconds. This is the smallest time period that can be measured by the CLOCK function. For a particular host-target configuration, a more precise real-time clock can be acquired from a vendor and this clock can be used for timing results. Calling this clock will require a separate interface and this will result in making the benchmarks non-portable. An imprecise clock can cause errors in benchmark timing measurements. Generally the execution time of a Ada feature is much smaller than SYSTEM.TICK.

2. **Clock Overhead:** Another problem is the inconsistent time required for the CLOCK function. Some compiler implementation return an aggregate data structure and this may require the calling of storage management functions. This may result in inconsistent timing for the CLOCK function. A program is used that calls upon the CLOCK function repeatedly. This measurement allows an assessment of measurement stability and overhead.

3. **Clock Jitter:** Clock readings are subject to the usual statistical variations associated with physical measurements and can be expected to show random variations known as jitter.

To overcome these problems, a technique known as the dual loop technique is used to measure the execution time for a specific feature.

3.1.1 Dual loop Benchmarks

The strategy used in measuring the execution time for a specific feature is known as the dual loop technique. In this technique an operation is performed repetitively, and the aggregate of multiple executions is timed. By performing the operation repetitively, the time duration of a test is increased and the system clock can measure this time precisely. In fact, this is done twice, once in a control loop without the feature being measured, once in a test loop with the feature. Subtracting the execution time of these two loops, and dividing by the number of executions yields a calculated time for one execution of the feature. The dual loop technique solves a number of problems that have been mentioned before.

1. By performing the operation repetitively the time duration of a test is increased and the system clock can measure this test precisely. By increasing the number of times the test problem is executed before accepting a measurement, the error in each estimate can be reduced.
2. Clock jitter compensation is achieved by executing each measurement for a minimum elapsed time. This time period is long enough so that the number of clock ticks will average out the random jitter.

For the dual loop strategy to be successful, it is essential that the control loop and the test loop (when executed without the feature being measured) should take identical amounts of execution time. Also, the time taken in calling the CLOCK function should remain constant. This assumption may always not be true and is discussed in detail in later sections.

3.1.1.1 Isolation of Features and Preventing Code Optimization

A major problem in dual loop benchmarks is the isolation of the feature that has to be measured. The benchmark should be designed such that the timing obtained after subtracting the control loop from the test loop measures ONLY the performance of the feature in question. Optimizations performed by a compiler can skew the benchmark results even to the point of rendering them totally incorrect. These optimizations (like removing code from test loops, eliminating subprogram calls, constant folding etc.) are performed by the compiler even if the optimize option is not specified at compile-time.

In dual loop benchmarks it is necessary to employ techniques that thwart optimizations by a compiler. This can be done by hiding constant variables from view, preventing simplification of loop constructs, and by arranging the order of compilation

for similar purposes.

1. Eliminate Constants and Expressions in Loops: For the control loop, some optimizing compilers may keep loop variables in registers (and not for the test loop). This will make the timing measurements erroneous as the test loop will be slower than it really is. To prevent this situation, the timing loop code contains a call on an external procedure and the number of loop iterations is controlled by external variables. Benchmarks have been designed such that there are no constants or expressions in the loops whose times are being measured.

2. Some compilers can generate different code for a construct depending on the nesting level where the construction occurred. If FOR loops are used to control the timing loops, a compiler might keep the innermost FOR loop index in a register. The time it takes to enter a FOR loop will depend on the nesting level of the FOR loops, because the nested loops must save and restore the registers for the outer loops. To prevent compiler optimizations, a while statement is used in both the control and test loops. The form of the while statement is:

   while I < N loop

   where I is the index variable and N is the iteration variable. A procedure is used to increment the index variable. The body of this is defined in the body of a separate library package. The iteration variable is defined and initialized in the specification of the same library package. The body of this library package is compiled separately from the package specification (with the body being compiled after the benchmarking unit). The purpose of keeping iteration values in variables (not constants), hiding the increment procedure in the body of the library package and separate compilation was to prevent the removal of benchmark loops by optimization.

3. Ensuring the Execution of the Feature being tested and preventing the elimination of the control loop by the compiler: To prevent this from happening, additional functions are inserted in the control and test loops and the feature being measured is placed in a subprogram which is called from a library unit.

   The form of the test loop is:

```
T1:= CLOCK;
while I < N loop
  control functions;
  DO_SEPARATE_PROC_F; --fn F whose time is measured
  INCREMENT(I);
end loop;
T2:= CLOCK;
Tm:= T2 - T1;
```

The form of the control loop is:

```
T1:= CLOCK;
while I < N loop
   control functions;
   DO_SEPARATE_PROC_NULL;
   INCREMENT(I);
end loop;
T2:= CLOCK;
Tm:= T2 - T1;
```

The bodies of the subprograms DO_SEPARATE_PROC_F and DO_SEPARATE_PROC_NULL are compiled separately and after the benchmarking unit. This prevents the compiler from removing anything from the control and test loops.

3.1.2 Measurement Accuracy

Once a feature has been identified, the next part is writing benchmarks that measure that feature accurately. In the University of Michigan report [1], it has been proved that the accuracy with which a feature can be measured depends on SYSTEM.TICK divided by the number of iterations of the benchmark. Most of the Michigan tests have a iteration count of 10000. If SYSTEM.TICK is 10 milliseconds, the accuracy of a measurement is within a microsecond. In the benchmarks that we have designed, the desired accuracy can be specified by adjusting the number of iterations.

3.1.3 Factors Affecting Benchmark Results

This section briefly describes the factors that may cause Ada benchmarks to produce incorrect results. In a recent report published by the Software Engineering Institute [2], and from previous efforts in running the University of Michigan benchmarks on DDC-I Ada Compiler System (hosted and targeted for the MicroVAX II), some negative results were encountered in running these benchmarks. If the control loop is a subset of the test loop, then the timing difference between the test loop and the control loop has to be positive. The SEI report has discussed in great detail the reasons that might cause the benchmarks to report negative results. Some the reasons that may cause erroneous results are:

- placement of code into memory
- asymmetrical translation (where the sequence has fewer machine code instructions)
- system software effects
- speed of main memory and use of cache memory
- Processor design: pipelining, multi-processor and distributed architectures.

Hence before dual loop benchmarks are run on a system, it is necessary to verify that the loop times are similar by coding identical loops in a procedure and comparing their execution times (calibration tests). Tests have been provided that measure the timing of identical loops. For systems with cache memory, it is essential to ensure that both the control and test loops are aligned to cause the same number of cache triggerings. A number of calibration tests that should be run before the benchmarks are executed. If the calibration tests show discrepancies, then this will require the examination of machine code and the target hardware.

3.1.4 Dual Loop Calibration Benchmarks

The purpose of the dual loop calibration benchmarks (loop_verify.a) is to determine if identically coded loops have identical execution times.

loop_verify.a: The benchmark design involves calling five functions executed in succession. Each function returns a DURATION value. This value is the time taken by a loop statement whose format is as follows:

```
START_TIME := CLOCK;
while OUTER_INDEX < OUTER_ITERATIONS loop
    INNER_INDEX := 0;
    while INNER_INDEX < INNER_ITERATIONS loop
        declare
        begin
            DO_NOTHING_IN_OUT(DUMMY1);
        end;  --declare
            INCREMENT(INNER_INDEX);
    end loop;
    INCREMENT(OUTER_INDEX);
end loop;
END_TIME := CLOCK;
```

The loop is executed (INNER_ITERATIONS * OUTER_ITERATIONS) times. The function calls are made in an arbitrary order to allow detection of any effects relating to the total number of machine cycles as opposed to the ordering of the loop routines. Each function call sequence is executed TEST_REPETITIONS times and there are five total sequences.

The benchmark output consists of the loop timings for each of the five function calls for each sequence. For each sequence run, the timings for the loops are listed and this is repeated TEST_REPETITIONS times. After the first sequence has been run, the second sequence follows the same pattern as above and is repeated TEST_REPETITIONS times.

The loop timings for each of the five function calls (irrespective of the sequence in which the loops are called) should be identical with an error margin of not more than 5% in the timing difference recorded. If the error margin for any loop timing is more than 5%, then the cause for the difference has to be investigated with the compiler vendor.

Verdix: The results of running this benchmark on the Verdix compiler indicated that the timings of the loop varied from 30.2 microseconds to 31.1 microseconds.

Interpretation of Results:

1. For the Verdix compiler, the difference of the timing values obtained is well within the 5% range of error. Hence, the dual loop technique can be applied to the Verdix compiler.

3.2 Ada Features For Microscopic Benchmarking

To identify Ada language and runtime features important for embedded systems applications is an extremely complex task. A list of Ada features whose performance measurement is necessary for designers of real-time embedded systems is presented below. Emphasis has been placed on Ada features used for real-time programming rather than on Ada software engineering principles like packages and generics.

- Tasking
- Memory management
- Exception handling
- Chapter 13 Benchmarks
- Interrupt Handling
- CLOCK overhead and Type Duration
- Numeric Computations
- Subprogram call overhead
- Pragmas
- Input/Output

Scheduling and delay statement is covered under Runtime implementation dependencies. There are benchmarks for all the major Ada language features that are important for real-time embedded systems. Since optimizing compilers generate different code for the same feature depending on the context in which the feature occurs, it has been attempted to benchmark a particular feature under different scenarios. The results will demonstrate the range of performance associated with a language feature. Examples include: subprogram calling, task creation, activation, and termination, task rendezvous, exception handling, I/O, numeric processing etc.

3.3 Tasking

For Ada to fulfill its potential for embedded systems, its model of concurrency - the tasking model - must be sufficiently fast to meet the timing needs of such systems. There is a significant amount of necessary overhead involved with tasking because status must be maintained for context switching, task dependencies, task abortion and termination, and task communication and synchronization.

Concern over efficiency and semantics of Ada tasking could force many

organizations using Ada to avoid the tasking facilities entirely, relying instead on a separately written executive. Not only will this defeat the whole purpose of Ada, but it will also produce inefficient code at the Ada source level, since Ada tasking features could be better suited for programming real-time embedded systems applications.

3.3.1 Tasking Benchmarks

Tasking overhead affects the efficiency of the system in both sizing and timing as the Ada runtime system contains the code that implements the Ada tasking features (entry calls, accepts, selects, etc.). The reason why benchmarking tasking constructs is extremely important is the fact that the Ada Language Reference Manual outlines the interface to the tasking system from an applications program and a method of communication and synchronization between tasks, but has left a large part of the implementation of that system undefined. For real-time embedded systems, it is essential that the timing overhead due to tasking constructs like task allocation, task activation/termination, task switching, synchronization and rendezvous be determined for a Ada compiler system's suitability for a real-time embedded application.

3.3.1.1 Task Activation/Termination

It is important to understand some of the activities that take place during task activation. Unlike other languages such as Modula-2 and HAL/S, which require the programmer to make explicit activation calls, Ada has a predefined activation facility. Ada elaboration starts with packages "withed" by the main procedure to find the leaves of the elaboration tree (lowest level unit with no other "with" dependencies). When the elaboration facility finds the lowest unit it:

1. creates a task control block for each task object, getting the space from the heap
2. creates task-stacks for each task body it elaborates, getting the space from the heap
3. allows tasks to start executing when the begin statement of the parent unit is reached. For task objects created from an access type, the elaboration and activation occur when the task object is created.

This process is repeated until all 'withed" packages are elaborated. Finally, the tasks declared in the main program are activated. Task activation requires some degree of synchronization with the parent unit, and some checks need to be performed. The start and end of task activation have to be synchronized with the parent unit or with the unit which causes the task activation. Furthermore, due to the semantics of the abort statement, a check must be made to see if the task has become abnormal.

The act of task termination, in general, requires some form of synchronization with certain other tasks (in simple cases a task termination must coincide with the termination of all dependent tasks). Typically the runtime system updates a database of terminate dependencies each time the runtime scheduler is invoked.

Some points to note about task activation/termination benchmarks are:

1. The time to elaborate, activate and terminate a task is measured as one value. The individual components of the measurements are too quick to measure with the available CLOCK resolution.
2. An important criteria for tasking benchmarks is the STORAGE_SIZE used by the tasks that are elaborated. Some implementations may implicitly deallocate the task storage space on return from a procedure or on exit from a block statement (when the task object is declared in that procedure or block statement). If task space is implicitly deallocated, the number of iterations can be increased to get greater accuracy for task activation/termination measurement. If task space is not deallocated on return from a procedure or block statement, then the attribute TASK_TYPE'STORAGE_SIZE can be changed such that the number of iterations (and hence the number of tasks activated and accuracy of the measurement) can be increased. The number of iterations is coarsely equal to the size of the heap space available divided by the STORAGE_SIZE specified.
3. The default task stack size for the Verdix Ada compiler is 10240 bytes. The Verdix Runtime System does not automatically deallocate objects (both listed in the Verdix documentation as well as determined by the benchmarks). Since the RAM available was 2 megabytes, this translates to about 175 tasks (=> 175 iterations) that could be activated to get the timing for task activation/termination. To get a better resolution for task activation/termination timings, the STORAGE_SIZE was reduced to 1000 bytes (via representation clause) and the number of iteration thus increased to 1000.

Table 1 lists the benchmarks that have been developed for Task activation/termination.

TABLE 1
Task Activation/Termination Benchmarks
(Verdix Execution Time in milliseconds)

File Name	Benchmark Description	Time
t00001.a	Task type in main, object in block statement	4.8
t00001_1	Task object is declared directly in block statement	4.8
t00001_2.a	Task type and object defined in package procedure	4.8
t00001_3.a	Task type in package, object in package procedure	4.8
t00001_4.a	Task type and object are declared in another task	4.8
t00002.a	Task type and array elaborated in a procedure	12.03
t00002_1.a	Task type in package, array in procedure	12.66
t00002_2.a	Task type in main, array in package procedure	12.04
t00003.a	Task object is declared as part of record	4.9
t00004.a	Task access type in main, task created via new	4.5
t00004_1.a	Task access type in block, task created via new	4.9
t00004_2.a	Task access type in main, array created via new	4.55
t00005.a	Task object in block statement, idle tasks=1	4.8
t00005_1.a	Task object in block statement, idle tasks=5	4.8
t00005_2.a	Task object in block statement, idle tasks=10	4.9
t00005_3.a	Task object in block statement, idle tasks=20	4.9
t00006.a	Task created via new allocator, idle tasks=1	4.5
t00006_1.a	Task created via new allocator, idle tasks=5	4.5
t00006_2.a	Task created via new allocator, idle tasks=10	4.6
t00006_3.a	Task created via new allocator, idle tasks=20	4.6

3.3.1.1.1 BENCHMARK: Measure task activation and termination time (without the new operator) where

- *t00001.a: Task type is declared in the main program and task object is declared in a block statement in the main program.*
  Entry into the block statement elaborates and activates the task object and the block statement exits when the task terminates.
  Verdix: Task elaborate, activate, and terminate time is 4.8 milliseconds.
- *t00001_1.a: Task object is declared directly in a block statement. The task object is declared directly and does not belong to any task type.*
  Entry into the block statement elaborates and activates the task object and the block statement exits when the task terminates.

Verdix: Task elaborate, activate, and terminate time is 4.8 milliseconds.

- *t00001_2.a: Task type and task object are defined in a procedure which is declared in a package.*
  This procedure is called from the main program. Entry into the procedure activates the task object and the procedure exits when the task terminates.
  Verdix: Task elaborate, activate, and terminate time is 4.8 milliseconds.

- *t00001_3.a: Task type is declared in a package and task object is declared in a procedure.*
  Procedure is declared in the same package in which the task type is declared. This procedure is called from the main program. Entry into the procedure activates the task object and the procedure exits when the task terminates.
  Verdix: Task elaborate, activate, and terminate time is 4.8 milliseconds.

- *t00001_4.a: Task type and task object are declared in another task which is declared in the main program.*
  This benchmark measures task activation and termination times for a task which is declared in another task.
  Verdix: Task elaborate, activate, and terminate time is 4.8 milliseconds.

Interpretation of results:

1. The activation and termination time of tasks for the various scenarios that are described above determine if a real-time programmer should declare tasks for time-critical modules in a) packages or in the main procedure, b) in procedures that are repeatedly called by other procedures, or c) within other tasks in the system.
2. The task activation/termination timings should be compared to the timings obtained by running the benchmarks on other compiler systems.

3.3.1.1.2 BENCHMARK: Measure activation/termination time for a) an array of tasks and b) task object declared as part of a record.
For array of tasks, the times recorded are of array'LENGTH task activations/terminations.

- *t00002.a: Task type and array object are defined and elaborated in a procedure which is declared in a package.*
  This procedure is called from the main program. Entry into the procedure activates the array of tasks and the procedure exits when the tasks terminate. The task STORAGE_SIZE may have to be reduced in order for a larger array to be activated, thus increasing the measurement resolution.
  Verdix: Task elaborate, activate, and terminate time is 12.03 milliseconds.

- *t00002_1.a: Task type is defined in a package. Task array object are defined in a procedure which is declared in the same package as the task type declaration.*
  Verdix: Task elaborate, activate, and terminate time is 12.66 milliseconds.

- *t00002_2.a: Task type is defined in the main program. Task array is defined and used in a procedure declared in the main program.*
  Verdix: Task elaborate, activate, and terminate time is 12.04 milliseconds.
- *t00003.a: Task object is declared as part of a record. Task and record type are declared in the main program.*
  Verdix: Task elaborate, activate, and terminate time is 4.9 milliseconds.

Interpretation of results:

1. For the Verdix compiler, the average time for task activation/termination for tasks declared in arrays is around 12.00 milliseconds, which is significantly higher than the task activation/termination time (4.8 milliseconds) for tasks declared in the main program. This is due to the fact that as each task in the array is elaborated, the task space for that task is left intact till all tasks in the array have been elaborated. Storage allocation times for tasks may deteriorate as more and more space has been allocated.
2. Timings for task object declared in a record should be compatible with simple task activation/termination timings (Section 3.3.1.1.1). This is due to the fact that the runtime system treats an Ada task object which is declared as the component of a record the same as a task object declared separately in the main program. For the Verdix compiler, the time for task declared in a record is 4.9 milliseconds as compared to 4.8 milliseconds for tasks declared in the main program.

3.3.1.1.3 BENCHMARK: Measure the time to activate and terminate a task created via the new allocator.

Since task access object does not exist on exit from the block statement, the timing measured includes both allocation and deallocation timings for the task as well as task activation and termination times.

- *t00004.a: Task type is declared in the main program, and task is created via the new allocator in a block statement.*
  The task STORAGE_SIZE may have to be reduced in order for a larger number of tasks to be created, thus increasing the measurement resolution.
  Verdix: Task elaborate, activate, and terminate time for this scenario via the new allocator is 4.5 milliseconds.
- *t00004_1.a: Task type is declared in a block and task object is created via the new allocator in the same block statement where the task was declared.*
  Verdix: Task elaborate, activate, and terminate time for this scenario via the new allocator is 4.9 milliseconds.
- *t00004_2.a: Task type is declared in the main program and an array of 1000 tasks is allocated in a block statement via the new statement.*
  Verdix: Task elaborate, activate, and terminate time for this scenario via the

new allocator is 4.55 milliseconds.

Interpretation of results:

1. The overhead associated with creating a task via the new operator can increase significantly if space is not implicitly deallocated when a task terminates upon exit from the block statement.
2. The Verdix compiler takes less time (4.5 milliseconds) for task activation/termination timing via the new allocator as compared to task objects declared in the main program (4.8 milliseconds).

3.3.1.1.4 BENCHMARK: Measure the time to activate and terminate a task object declared in the declarative part of a block as the number of existing active tasks keeps on increasing.

This benchmark measures the degradation in task activation/termination time as the number of tasks in the system increases. A task activation could involve a stack allocation, heap allocation, or specific allocation from a pre-allocated storage collection. As more and more tasks are created, task activation time may increase due to the possible increase in storage allocation time.

- *t00005.a : Number of existing active tasks = 1.*
  Verdix: Task elaborate, activate, and terminate time for this scenario is 4.8 milliseconds.
- *t00005_1.a:Number of existing active tasks = 5.*
  Verdix: Task elaborate, activate, and terminate time for this scenario is 4.8 milliseconds.
- *t00005_2.a:Number of existing active tasks = 10.*
  Verdix: Task elaborate, activate, and terminate time for this scenario is 4.9 milliseconds.
- *t00005_3.a:Number of existing active tasks = 20.*
  Verdix: Task elaborate, activate, and terminate time for this scenario is 4.9 milliseconds.

Interpretation of results

1. These timings are compared with the timing obtained in Section 3.3.1.1.1 (t00001.a) to determine degradation in task activation/termination timings (if any) as the number of active tasks in the system increases. For the Verdix compiler, task activation/termination time increases to 4.9 milliseconds from 4.8 milliseconds when the number of tasks that are active is 10 or more.

3.3.1.1.5 BENCHMARK: Measure the time to activate and terminate a task created via the new allocator in a block as the number of existing active tasks

keeps on increasing.
This benchmark measures the degradation in task activation/termination time as the number of tasks in the system increases. Task allocation time will depend on the implementation scheme and possibly on the history of previous allocations at the time of activation. As more and more tasks are created, task activation time may increase due to the possible increase in storage allocation time.

- *t00006.a : Number of existing active tasks = 1.*
  Verdix: Task elaborate, activate, and terminate time for this scenario via the new allocator is 4.5 milliseconds.
- *t00006_1.a:Number of existing active tasks = 5.*
  Verdix: Task elaborate, activate, and terminate time for this scenario via the new allocator is 4.5 milliseconds.
- *t00006_2.a:Number of existing active tasks = 10.*
  Verdix: Task elaborate, activate, and terminate time for this scenario via the new allocator is 4.6 milliseconds.
- *t00006_3.a:Number of existing active tasks = 20.*
  Verdix: Task elaborate, activate, and terminate time for this scenario via the new allocator is 4.6 milliseconds.

Interpretation of results

1. These timings are compared with the timing obtained in Section 3.3.1.1.3 (t00004.a) to determine degradation in task activation/termination timings (if any) as the number of active tasks in the system increases. For the Verdix compiler, task activation/termination time increases to 4.6 milliseconds from 4.5 milliseconds when the number of tasks that are active is 10 or more.

3.3.1.2 Task Synchronization

In Ada, tasks communicate with each other via the rendezvous mechanism. Rendezvous are effectively similar to procedure calls, yet they are much more complex to implement, and therefore create a tremendous amount of overhead for the runtime system. This overhead affects the efficiency of the system in both sizing and timing. One task must always wait for the other to reach the point of the rendezvous, the system must invoke the rendezvous when both tasks are ready, and context switches are required between the tasks during rendezvous. Priorities are not static during a rendezvous and this presents additional overhead during execution time.

The calling task is suspended until the rendezvous is completed. The called task then evaluates the statement in the range of the accept statement. Rendezvous involves at least two context switches: one to the runtime system and then another to the acceptor if it is ready to accept the rendezvous. Before control is transferred to the acceptor of the rendezvous, additional overhead is involved in checking if the acceptor is indeed ready to conduct the rendezvous. Because of the timing constraints in a real-time embedded system, it is essential that the rendezvous mechanism be as efficient as possible. Time for rendezvous can degrade with the number of eligible tasks due to the search and sorting involved with prioritized scheduling.

In a nutshell, task rendezvous involves:

- Passing rendezvous parameters which may involve both the task's stacks or the allocation of a separate area for passing large structures. Returned rendezvous parameters may involve the copying of data from one task's data area to the other's.
- Performing the appropriate constraint checking
- Determining the availability of space.

Table 2 lists the benchmarks for simple rendezvous.

TABLE 2
Simple Rendezvous Benchmarks (No Parameters P ssed)
(Verdix Execution Time in microseconds)

File Name	Benchmark Description	Time
r00001.a	Procedure calls entry of task declared in main	358
r00001_1.a	Procedure calls entry in task created via new	356.8
r00001_2.a	Main calls entry in task decl in package	355.2
r00002.a	Main calls two entries in two tasks decl in package	354.3
r00002_1.a	Main calls 10 entries in ten tasks decl in package	353.9
r00002_2.a	Main calls 10 entries in one task decl in package	352.6
r00003.a	Main calls 1st entry in select, 2 entries decl	412.7
r00003_1.a	Main calls last entry in select, 2 entries decl	436.8
r00003_2.a	Main calls 1st entry in select, 10 entries decl	474.7
r00003_3.a	Main calls last entry in select, 10 entries decl	684.3
r00003_4.a	Main calls 6th entry in select, 10 entries decl	581.0
r00003_5.a	Main calls 1st entry in select, 20 entries decl	553.5
r00003_6.a	Main calls last entry in select, 20 entries decl	994.1
r00003_7.a	Main calls 11th entry in select, 20 entries decl	786.3
r00004.a	Main calls 1st entry out of 2, 1st guard true next false	414.3
r00004_1.a	Main calls last entry out of 2, 1st guard false next true	456.7
r00004_2.a	Main calls 1st entry out of 20, 1st guard true rest false	563.6
r00004_3.a	Main calls last entry out of 20, last guard true rest false	1337.2
r00004_4.a	Main calls 11th entry out of 20, 11th guard true rest false	792.1
r00004_5.a	Main calls 11th entry out of 20, all guards true	786.3

3.3.1.2.1 BENCHMARK: Measure time for simple rendezvous
The simple rendezvous time gives a lower bound on the rendezvous time because no extraneous units of execution are competing for the CPU. This overhead is expected to occur each time two tasks are in a rendezvous and does not include any execution time for the statements within the accept body.

- *r00001.a: Procedure in main program calls an entry of task which is declared in main program. The entry call is made with no parameters.*
  Verdix: Task rendezvous time for this scenario is is 358 microseconds.
- *r00001_1.a: Procedure in main program calls an entry in a task, where the access type is declared in a package. The task is allocated via the new operator in a block statement in the main program. The entry call is made with no*

parameters.
Verdix: Task rendezvous time for this scenario is is 356.8 microseconds.

- *r00001_2.a: Main program calls an entry in another task where the task object is declared in a library package. Entry call is made with no parameters.*
  Verdix: Task rendezvous time for this scenario is is 355.2 microseconds.

Interpretation of results:

1. Task rendezvous time for the various scenarios discussed above vary from 355 to 358 microseconds. This can be compared to rendezvous times obtained by running the benchmarks on another compiler. Large task rendezvous time may force real-time embedded system programmers not to use task rendezvous in their software development phase or to use another compiler whose rendezvous times are acceptable.

3.3.1.2.2 BENCHMARK: Measure time for simple rendezvous. More than one entry is called to measure rendezvous time. These entries can all be in a single task or single entries in multiple tasks.
The results of these tests will indicate if it is advantageous to have more tasks with less entries or less tasks with more entries. Rendezvous times obtained can be compared to rendezvous times obtained by benchmarks in Section 3.3.1.2.1. These times should be compatible and any significant difference in these times should be investigated further with the compiler vendor.

- *r00002.a: Main procedure calls two entries in two tasks declared in a package. The entry calls are made with no parameters. Each task has a single entry declared.*
  This scenario is essentially the time for simple rendezvous with two entries in two separate tasks.
  Verdix: Task rendezvous time for this scenario is is 354.3 microseconds.
- *r00002_1.a: Main procedure calls 10 entries in ten tasks declared in a package. The entry calls are made with no parameters.*
  Verdix: Task rendezvous time for this scenario is is 353.9 microseconds.
- *r00002_2.a: Main procedure calls 10 entries declared in a single task with no parameters.*
  The entry calls are made one after the other and the accept statements are also sequential and not in a select statement.
  Verdix: Task rendezvous time for this scenario is is 352.6 microseconds.

Interpretation of results:

1. For the Verdix compiler, the timing for rendezvous is nearly the same for the scenarios in which the main program calls ten entries in 10 different tasks or the main program calls 10 entries in one task.

3.3.1.2.3 BENCHMARK: Measure the effect on the time required for a simple rendezvous, where a procedure in the main program calls an entry in another task with no parameters as the number of accept alternatives in the selective wait increases.

The times measured by these tests determine the effect on entry call time as the number of accept alternatives in a select statement increases. For some implementations, time for a rendezvous may also be affected by the position of the accept alternative in the select statement. Based on these tests, application designers can choose to place the most time-critical accept statements in a certain manner.

This benchmark is executed with the following scenarios:

- *r00003.a (r00003_1.a): Main procedure calls first (last) entry in a select statement. 2 entries declared with no parameters.*
  Verdix: Task rendezvous time for r00003.a (r00003_1.a) is 412.7 (436.8) microseconds.
- *r00003_2.a (r00003_3.a, r00003_4.a): Main procedure calls first (last, 6th) entry in a select statement. 10 entries declared with no parameters.*
  Verdix: Task rendezvous time for r00003_2.a (r00003_3.a, r00003_4) is 474.7 (684.3, 581.0) microseconds.
- *r00003_5.a (r00003_6.a, r00003_7.a): Main procedure calls first (last, 11th) entry in a select statement : 20 entries declared with no parameters.*
  Verdix: Task rendezvous time for r00003_5.a (r00003_6.a, r00003_7) is 553.5 (994.1, 786.3) microseconds.

Interpretation of results:

1. For the Verdix compiler, the measurements taken indicate that the more the number of entries in a select statement, the more time it takes to rendezvous with any entry in the select statement.
2. Also, for the Verdix compiler the later the position of the accept in the select statement, the more time it takes for the rendezvous to complete.

3.3.1.2.4 BENCHMARK: Measure the effect of guards (on accept statements) on rendezvous time, where the main program calls an entry in another task (with no parameters) as the number of accept alternatives in the select statement increases.

The times measured by these tests determine the affect on rendezvous time of guard statements as the number of accept alternatives in a select statement increases. For some implementations, rendezvous time may depend on the number of open guard statements.

This benchmark is executed with the following scenarios:

- *r00004.a (r00004_1.a): Main program calls first (last) entry in select statement. 2 entries are declared: first guard is true (false), next one is false (true).*

Verdix: Task rendezvous time for r00004.a (r00004_1.a) is 414.3 (456.7) microseconds. The task rendezvous time for the entry call made to the first accept statement in the select statement when the first guard is true is 414.3 microseconds as compared to 412.7 microseconds without the guard condition. The task rendezvous time for the entry call made to the last accept statement in the select statement when the last guard is true is 456.7 microseconds as compared to 436.8 microseconds without the guard condition.

- *r00004_2.a (r00004_3.a, r00004_4.a): Main program calls first (last, 11th) entry in select statement. 20 entries are declared: first (last, 11th) guard is true, rest false.*
  Verdix: Task rendezvous time for r00004_2.a (r00004_3.a, r00004_4) is 563.6 (1337.2, 792.1) microseconds. The task rendezvous time for the entry call made to the first accept statement in the select statement when the first guard is true is 563.6 microseconds as compared to 553.5 microseconds without the guard condition. The task rendezvous time for the entry call made to the last accept statement in the select statement when the last guard is true is 1337.2 microseconds as compared to 994.1 microseconds without the guard condition. The task rendezvous time for the entry call made to the 11th accept statement in the select statement when the 11th guard is true is 792.1 microseconds as compared to 786.3 microseconds without the guard condition.

- *r00004_5.a: All guards are true and 11th entry is called.*
  Verdix: The task rendezvous time for the entry call made to the 11th accept statement in the select statement when all guards are true is also 792.1 microseconds as compared to 786.3 microseconds without the guard condition.

Interpretation of results:
Some conclusions that can be drawn about the Verdix compiler are as follows:

1. There is non-trivial time involved in evaluating guards for the accept statements. Rendezvous time with guards is more than rendezvous time without guards.
2. The greater the number of guards in the select statement, and the later the position of the accept statement in the select, the more time it is going to take for the rendezvous.

Table 3 lists the benchmarks for complex rendezvous. For each benchmark, the direction passed for the parameters, number and type of parameters, and the size of the parameters is also listed in the table. For array parameters, the column **Size** denotes the attribute LENGTH of the array.

TABLE 3
Complex Rendezvous Benchmarks
(Verdix Execution Time in microseconds)

File Name	Direction Passed	Type and Number Passed	Size	Time
r00005_i.a	In	Integer Array	1	376.5
r00005_o.a	Out	Integer Array	1	399.1
r00005_io.a	In Out	Integer Array	1	397.4
r00005_1_i.a	In	Integer Array	1000	355.7
r00005_1_o.a	Out	Integer Array	1000	355.5
r00005_1_io.a	In Out	Integer Array	1000	355.7
r00005_2_i.a	In	Integer Array	10000	354.5
r00005_2_o.a	Out	Integer Array	10000	354.5
r00005_2_io.a	In Out	Integer Array	10000	354 5
r00005_3_i.a	In	1 Integer		354.9
r00005_3_o.a	Out	1 Integer		354.6
r00005_3_io.a	In Out	1 Integer		355.4
r00005_4_i.a	In	10 Integers		361.9
r00005_4_o.a	Out	10 Integers		367.6
r00005_4_io.a	In Out	10 Integers		376.4
r00005_5_i.a	In	100 Integers		455.8
r00005_5_o.a	Out	100 Integers		546.0
r00005_5_io.a	In Out	100 Integers		543.7

3.3.1.2.5 BENCHMARK: Measure the time required for a complex rendezvous, where a procedure in the main program calls an entry in another task with different type, number and mode of the parameters.
Rendezvous time may depend on the size and type of the passed parameters which may involve both the task stacks or the allocation of a separate area for passing large structures. Returned rendezvous parameters may involve the copying of data from one task's data area to the other's. Also the appropriate constraint checking has to be done while passing the parameters. Increasing rendezvous times for array parameters as the size of the array increases implies that the implementation uses pass by copy instead of pass by reference.
The types and modes of the parameters are as follows:

- *r00005_i.a (r00005_o.a, r00005_io.a): Main calls an entry in another task with Integer array of size 1 as parameter: mode in (out, in out).*
  Verdix: Rendezvous time for r00005_i.a (r00005_o.a, r00005_io.a) is 376.5

(399.1, 397.4) microseconds.

- *r00005_1_i.a (r00005_1_o.a, r00005_1_io.a): Main calls an entry in another task with Integer array of size 1000 as parameter: mode in (out, in out).*
  Verdix: Rendezvous time for r00005_1_i.a (r00005_1_o.a, r00005_1_io.a) is 355.7 (355.5, 355.7) microseconds.
- *r00005_2_i.a (r00005_2_o.a, r00005_2_io.a): Main calls an entry in another task with Integer array of size 10000 as parameter: mode in (out, in out).*
  Verdix: Rendezvous time for r00005_2_i.a (r00005_2_o.a, r00005_2_io.a) is 354.5 (354.5, 354.5) microseconds.
- *r00005_3_i.a (r00005_3_o.a, r00005_3_io.a): Main calls an entry in another task with one Integer parameter: mode in (out, in out).*
  Verdix: Rendezvous time for r00005_3_i.a (r00005_3_o.a, r00005_3_io.a) is 354.9 (354.6, 355.4) microseconds.
- *r00005_4_i.a (r00005_4_o.a, r00005_4_io.a): Main calls an entry in another task with 10 Integer parameters: mode in (out, in out).*
  Verdix: Rendezvous time for r00005_4_i.a (r00005_4_o.a, r00005_4_io.a) is 361.9 (367.6, 376.4) microseconds.
- *r00005_5_i.a (r00005_5_o.a, r00005_5_io.a): Main calls an entry in another task with 100 Integer parameters: mode in (out, in out).*
  Verdix: Rendezvous time for r00005_5_i.a (r00005_5_o.a, r00005_5_io.a) is 455.8 (546.0, 543.7) microseconds.

Interpretation of results:

1. The measurements indicate that the rendezvous time for passing an integer array of size 1 is significantly higher than rendezvous time for integer arrays of size 1000 or more.
2. As far as integer parameters are concerned, the Verdix compiler uses pass by copy (due to the fact that the time for rendezvous increases with the increase in the number of integer parameters). Also, the time for mode **out** and **in out** parameters is more than the time required for parameters of mode **in**. This is logical since the compiler has to copy back the change in value that can occur with a variable of type **out** or **in out**.

Table 4 lists more rendezvous benchmarks. All parameters in these benchmarks are passed with mode **in out**.

TABLE 4
More Rendezvous Benchmarks
(Verdix Execution Time in microseconds)

File Name	Benchmark Description	Time
r00006_1_1.a	1st entry out of 2 called with 10 integers	429.4
r00006_1_2.a	1st entry out of 2 called with 100 integers	692.5
r00006_2_1.a	Last entry out of 2 called with 10 integers	454.0
r00006_2_2.a	Last entry out of 2 called with 100 integers	614.2
r00006_3_1.a	1st entry out of 10 called with 10 integers	503.1
r00006_3_2.a	1st entry out of 10 called with 100 integers	653.1
r00006_4_1.a	Last entry out of 10 called with 10 integers	692.7
r00006_4_2.a	Last entry out of 10 called with 100 integers	944.6
r00006_5_1.a	1st entry out of 20 called with 10 integers	581.0
r00006_5_2.a	1st entry out of 20 called with 100 integers	730.7
r00006_6_1.a	Last entry out of 20 called with 10 integers	984.0
r00006_6_2.a	Last entry out of 20 called with 100 integers	1235.7
r00007.a	Overhead due to terminate alternative	0.9
r00008.a	Overhead of conditional entry call,rendezvous complete	0.6
r00008_1.a	Overhead of conditional entry call,rendezvous incomplete	25.3
r00009.a	Overhead of timed entry call,rendezvous complete	9.7
r00009_1.a	Overhead of timed entry call,rendezvous incomplete	25.3
r00011.a	Main calls an entry with 100 Integers,Idle tasks = 1	530.0
r00011_1.a	Main calls entry with 100 Integers,Idle tasks = 5	529.9
r00011_2.a	Main calls entry with 100 Integers,Idle tasks = 10	530.0
r00011_3.a	Main calls entry with 100 Integers,Idle tasks = 20	529.9

3.3.1.2.6 BENCHMARK: Measure the effect on time required for a complex rendezvous, where the main program calls an entry in another task with different type, number and mode of the parameters as the number of accept alternatives in the select statement increase.

For some implementations, time for a rendezvous may be affected by the position of the accept alternative in the select statement.

This benchmark is executed with the following scenarios:

- *r00006_1_1.a (r00006_1_2.a): First entry out of 2 called with 10 (100) integers:mode in out.*

  Verdix: Rendezvous time for r00006_1_1.a (r00006_1_2.a) is 429.4 (692.5)

microseconds.

- *r00006_2_1.a (r00006_2_2.a): Last entry out of 2 called with 10 (100) integers:mode in out.*
  Verdix: Rendezvous time for r00006_2_1.a (r00006_2_2.a) is 454.0 (614.2) microseconds.
- *r00006_3_1.a (r00006_3_2.a): First entry called in a select statement with 10 (100) integers: 10 entries.*
  Verdix: Rendezvous time for r00006_3_1.a (r00006_3_2.a) is 503.1 (653.1) microseconds.
- *r00006_4_1.a (r00006_4_2.a): Last entry called in a select statement with 10 (100) integers: 10 entries.*
  Verdix: Rendezvous time for r00006_4_1.a (r00006_4_2.a) is 692.7 (944.6) microseconds.
- *r00006_5_1.a (r00006_5_2.a): First entry called in a select statement with 10 (100) integers: 20 entries.*
  Verdix: Rendezvous time for r00006_5_1.a (r00006_5_2.a) is 581.0 (730.7) microseconds.
- *r00006_6_1.a (r00006_6_2.a): Last entry called in a select statement with 10 (100) integers: 20 entries.*
  Verdix: Rendezvous time for r00006_6_1.a (r00006_6_2.a) is 984.0 (1235.7) microseconds.

Interpretation of results:

1. For the Verdix compiler, the time for rendezvous call to the last entry with 100 integer parameters (mode in out) increases from 614 microseconds (2 entries) to 944 microseconds (10 entries) to 1235 microseconds (20 entries). Thus, it can be deduced that time for rendezvous with integer parameters increases linearly as the number of accept statements in the select statement increases.

3.3.1.2.7 BENCHMARK: Measure the cost of using the terminate option in a select statement.

A group of tasks (children of the same parent) can terminate by using the terminate option of the select statement. If the overhead due to the terminate option is high, then this option should not be used (especially if the selective wait is inside a loop).

r00007.a: To determine the cost of the terminate option in a select statement, a server task uses a guarded terminate in its select statement. The select statement was executed N times with the guard being false and N times with the guard being true. The task does not terminate as a result of this select statement. Therefore the additional time required when the terminate guard was true is the additional

overhead of the terminate option.
Verdix: The cost of using the terminate option in a select statement is 0.9 microseconds (Control:456.9, Test:457.8)

Interpretation of results

1. The cost of using the terminate option has an additional overhead of 0.9 microseconds each time the select statement is executed.

3.3.1.2.8 BENCHMARK: Measure the overhead due to a conditional entry call when a) the rendezvous is completed (*r00008.a*) and b) the rendezvous is not completed (*r00008_1.a*).
r00008.a, r00008_1.a: Two tasks are used: a calling task and a server task. To measure the time when the rendezvous did not complete, the server task waited on one entry while the calling task made attempts to call a different entry. To measure the added overhead when the rendezvous did complete, a calling task using the conditional and timed entry calls was compared to a calling task using simple entry calls.
Verdix: Overhead due to conditional entry call when the rendezvous is completed is 0.6 microseconds. Overhead due to conditional entry call when the rendezvous is not completed is 25.3 microseconds.

Interpretation of results:

1. When one task wishes to call an entry in another task it has the option of making the call if and only if the called task is ready to accept the call or to wait until the caller is ready. The first of these two choices is a conditional entry call. A conditional entry call where the rendezvous is not completed was measured to take 25.3 microseconds. This is the time required to execute the else statement in the select.
2. The overhead due to the else statement when the rendezvous is completed is 0.9 microseconds. This is the runtime overhead of the else statement when it is not executed as the rendezvous takes place. Obviously, this time is much less than the time to execute the else statement (25.3 microseconds).
3. This overhead has to be considered whenever polling is used to establish synchronization between tasks.

3.3.1.2.9 BENCHMARK: Measure the overhead due to a timed entry call when a) the rendezvous is completed (*r00009.a*) and b) the rendezvous is not completed (*r00009_1.a*).
Like the conditional entry mechanism, the timed entry mechanism gives the calling task a degree of control over the call to the task entry. A timed entry call allows the calling task to specify how long it is willing to wait for the rendezvous

to start. If this time limit expires prior to the start of the rendezvous then the call is canceled.
r00009.a, r00009_1.a: Two tasks are used: a calling task and a server task. To measure the time when the rendezvous did not complete, the server task waited on one entry while the calling task made attempts to call a different entry. For the timed entry call a delay of 0.0 was used. To measure the added overhead when the rendezvous did complete, a calling task using the timed entry calls was compared to a calling task using simple entry calls.
Verdix: Overhead due to timed entry call when the rendezvous is completed is 9.7 microseconds. Overhead due to conditional entry call when the rendezvous is not completed is 25.3 microseconds.

Interpretation of results:

1. A timed entry call incurs a measured overhead of 25.3 microseconds when the delay expires, and 9.7 microseconds when the rendezvous takes place before the delay expires.

3.3.1.2.9 BENCHMARK: Measure the effect on time required for a complex rendezvous, where a procedure in the main program calls an entry as the number of activated tasks in the system increases.

- *r00011.a (r00011_1.a, r00011_2.a, r00011_3.a): Main program calls an entry in another task with 100 Integer parameters: Idle tasks = 1 (5, 10, 20).*
  Verdix: Time for r00011.a (r00011_1.a, r00011_2.a, r00011_3.a) is 530.0 (529.9, 530.0, 529.9) microseconds.

Interpretation of results:

1. Time for rendezvous can degrade with the number of eligible tasks due to the search and sorting involved with prioritized dispatching. For the Verdix compiler time for rendezvous remains the same for up to 20 idle tasks.

3.3.2 Remarks on Ada Tasking

There are specific concerns in the real-time application community regarding the semantics of the Ada tasking model and its potential implementation overhead. Some areas of concern emerge as a result of this and other benchmarking efforts [1]. A brief discussion of some of these problems is necessary to make the readers aware of the dilemma facing designers of real-time embedded applications:

- In many Ada compilers, the performance of Ada tasking model is divorced from performance concerns. In real-time embedded systems, performance is a critical part of the requirement. One should be able to predict the response time of a system in Ada. While one does not want to return to the cyclic executive with its attendant costs in code clarity and maintainability, some way must be found of predicting realistically the response time of a system engineered in Ada. Traditional approaches to real-time operating systems have relied on precise timing and tight control over the sequence and length of execution of individual system components. Ada tasking model does not support this type of control.
- Rendezvous abstractions are inherently inefficient. In measurements that have been done in this report, rendezvous time is at least 353 microseconds which is not acceptable in many real-time systems. Most of the overhead stems from all the queue operations needed to implement rendezvous. Rendezvous times are generally about 30 times that of a procedure call which is way too much for real-time embedded applications. Another source of inefficiency is the generality of the tasking constructs, i.e. the requirements on entry calls, accepts, time outs or synchronous termination of a set of tasks.
- For the translation of concurrency paradigms, an application may have to use intermediate tasks with the risk of compromising real-time performance. Also, the resulting code may not be very readable, for example when entry families are used to simulate prioritized queues.

Some solutions that have been proposed in order to achieve tasking efficiency include:

- Improved runtime implementation techniques that eliminate unused functionality
- Implementation by the compiler of various task optimizations so that many resource bound applications can use tasking in an efficient manner
- Proposals by the ARTEWG [6] so that a low-level of task control enables a real-time application sufficient degree of freedom.

3.4 Memory Management

Ada is the first high order language intended for mission critical, real-time applications that requires dynamic memory allocation and deallocation. The Ada language encompasses dynamic objects of unconstrained types, objects of access types, workspaces of tasks, compiler generated temporary objects for computation, and subprograms with locally defined data. Subprograms

can be called recursively, with more storage required at each level of recursion. The amount of storage required in these circumstances cannot be determined by static examination of a program and benchmarks must be executed to determine the efficiency of an implementation's storage utilization.

The quality of a program's runtime performance can depend greatly on the flexibility and precision with which storage allocation decisions are made (either by the programmer or, at runtime, by the runtime system). This means that Ada benchmarks should test for performance characteristics such as control over storage allocation. Control over storage allocation includes the ability to allocate blocks of storage for specified purposes, the ability to defer allocation until the amount of storage needed for each purpose is known, and the ability to control the deallocation of storage.

3.4.1 Storage Mechanisms in Ada

The memory management function of the Ada runtime system is responsible for allocation and deallocation of storage at runtime. The design of the runtime system affects the amount of storage used during subprogram calls, task activation, and dynamic allocation of storage. If the system runs out of storage, the memory management function raises the exception STORAGE_ERROR.

There are two main schemes for dynamic memory management in an Ada runtime environment: stack and heap storage schemes.

1. **Stack Structured Allocation Schemes:** Stack storage is allocated for a task as well as for a main program. Local variables of subprograms and tasks are allocated on the stacks assigned to the task to which the subprogram belongs. The memory management function of the runtime system allocates and deallocates space on the stack and checks for stack overflow. Not only local variables are placed on the stack, but administrative variables such as return addresses, lexical parent pointers, dependent task counters, exception scope information, etc. are included in the activation record of a subprogram. One common attribute of objects stored on the stack is that their lifetimes are nested.

2. **Heap Structured Allocation Schemes:** Lifetimes of objects created by Ada allocators cannot be determined at compile time and hence the heap storage mechanism is used to allocate storage for objects created by the new allocator. Storage overhead associated with dynamic allocation may be incurred when an access type is declared and again

each time a variable is allocated. This allocation/deallocation of space is managed by the memory management function.

A recent article published in Ada Letters, "Mapping Ada onto Embedded Systems:Memory Constraints" [14] has an extremely articulate discussion of the memory requirements of an Ada program while executing on an embedded target. That discussion is presented here for a better understanding of the stack and heap issues.

3.4.1.1 Memory Requirements of an Ada Program

A hypothetical Ada memory model, ignoring the needs of the runtime system, is discussed. There are often a few reserved locations set aside for hardware purposes. Most computers dedicate location 0 for a reset interrupt vector. Following locations may be dedicated for an interrupt vector table followed by a small monitor program that can be used for host-to-target downloading and debugging. The remaining memory can then be used for the application code and the associated Ada runtime system.

The ROM area will hold the Ada application and runtime code, constants, and initialization data. Following this is a DATA RAM for any global data that the linker may choose to pre-allocate or pre-elaborate. Any objects that exist for the lifetime of the entire program (static objects) can be pre-allocated or pre-elaborated. In addition, there may be a separate area of RAM for shared global memory, or memory mapped I/O devices. When address clauses locating objects at absolute addresses are used, it is important to place such objects in an area that the linker will not use for any other objects. For example, to place objects at absolute locations within the heap and stacks could be dangerous. Note that the ROM and RAM areas do not have to be contiguous while heap and stack areas must be contiguous.

Remaining higher memory will then contain stack and heap areas. When tasking is not present, an Ada program only requires a single heap for all allocators and a single stack for all procedure/function subprogram calls. The stack grows and shrinks as each subprogram is called, sequentially executes, and then completes. With tasking, each task can proceed independently as if it had its own CPU; therefore, each task needs its own stack. Whenever a task calls a subprogram, the subprogram will make use of the stack of the task that is calling it. Any subprograms subsequently called by that subprogram also make use of the task provided stack. The task_type'STORAGE_SIZE attribute is used to set aside storage space for each task's stack, typically called a task_stack. To figure out the minimum

size of the task_stack, find out how many parameters and local variables are used by each subprogram chain that the task will ever call. For recursive calls, scale up by the number of possible recursions. The size needed is determined by the calling chain that uses the most stack space.

On a single CPU system, tasks will take turns executing. Because each task will need its own stack, and because tasks need to be able to execute in any order, each task_stack will have to be allocated from a structure that accounts for random accesses, i.e. the heap. Dynamic creation of tasks by means of the new allocator is another reason for having task-stacks allocated from the heap.The LRM defines an Ada main program as a procedure that is invoked by an "environment task", hence the stack for an environment task could also come from the heap. However, an implementation may choose to use the system stack for the "environment stack" workspace.

There is at least one more use for the heap. Whenever the runtime system switches control from one task to the next, it must save most, if not all of the hardware registers. In addition, the runtime system might want to store miscellaneous information about a task (when it was last called, etc). Typically, this information is stored in the heap in a structure called the task control block.

3.4.2 Dynamic Memory Allocation Benchmarks

3.4.2.1 BENCHMARK: Measure time for allocating storage known at compile time.

In the first type of test cases, time is measured to allocate and deallocate a fixed amount of storage upon entering a subprogram or a declare block. The objects are declared in subprogram or declare blocks. The size of the objects is known at compilation time, but space for the objects is allocated on the stack at runtime. Different types and sizes of objects are allocated (as listed in Table 5). Times to allocate various numbers of types INTEGER and ENUMERATION are measured as well as the times to allocate various sizes of arrays, records, and STRINGs. The objective is to determine the allocation overhead involved and if there is any difference in the overhead based on the type of object allocated. Different types and sizes of objects are allocated (as listed in Table 5). In Table 5, the column **size of object** for STRINGs is specified as STRING'LENGTH, for integer arrays **size of object** is specified as array'length, and for records the **size of object** is specified as the number of fields in the record.

TABLE 5
Dynamic Allocation:Storage Allocated Is Fixed
(Verdix Execution Time in Microseconds)

File Name	Type Declared	Number Declared	Size of Object	Time
dd_in1.a	Integer	1		0.4
dd_in10.a	Integer	10		0.4
dd_in100.a	Integer	100		0.4
dd_st1.a	String	1	1	0.4
dd_st10.a	String	1	10	0.4
dd_st100.a	String	1	100	0.4
dd_en1.a	Enumeration	1		0.4
dd_en10.a	Enumeration	10		0.4
dd_en100.a	Enumeration	100		0.4
dd_ar1.a	Array of Integer	1	1	0.4
dd_ar10.a	Array of Integer	1	10	0.8
dd_ar100.a	Array of Integer	1	100	0.8
dd_ar1k.a	Array of Integer	1	1000	0.8
dd_ar10k.a	Array of Integer	1	10000	5.2
dd_ar100k.a	Array of Integer	1	100000	5.2
dd_rc1.a	Record of Integer	1	1	13.9
dd_rc10.a	Record of Integer	1	10	68.6
dd_rc100.a	Record of Integer	1	100	678.5

- *dd_in1.a (dd_in10.a, dd_in100.a): Measure time to allocate and deallocate 1 (10, 100) integers upon entering a subprogram.*
  Verdix: Time for dd_in1.a (dd_in10.a, dd_in100.a) is 0.4 (0.4, 0.4) microseconds.

- *dd_st1.a (dd_st10.a, dd_st100.a): Measure time to allocate and deallocate strings of length 1 (10, 100) upon entering a subprogram.*
  Verdix: Time for dd_st1.a (dd_st10.a, dd_st100.a) is 0.4 (0.4, 0.4) microseconds.

- *dd_en1.a (dd_en10.a, dd_en100.a): Measure time to allocate and deallocate 1 (10, 100) enumeration variables upon entering a subprogram.*
  Verdix: Time for dd_en1.a (dd_en10.a, dd_en100.a) is 0.4 (0.4, 0.4) microseconds.

- *dd_ar1.a (dd_ar10.a, dd_ar100.a, dd_ar1k.a, dd_ar10k.a, dd_ar100k.a): Measure time to allocate and deallocate an array of integer with 1 (10, 100, 1000, 10000, 100000) elements upon entering a subprogram.*
  Verdix: Time for dd_ar1.a (dd_ar10.a, dd_ar100.a, dd_ar1k.a, dd_ar10k.a, dd_ar100k.a) is 0.4 (0.8, 0.8, 0.8, 5.2, 5.2) microseconds.
- *dd_rc1.a (dd_rc10.a, dd_rc100.a): Measure time to allocate and deallocate an record with 1 (10, 100) integer fields upon entering a subprogram.*
  Verdix: Time for dd_rc1.a (dd_rc10.a, dd_rc100.a) is 13.9 (68.6, 678.5) microseconds.

Interpretation of results:

1. For the Verdix Compiler, time required to allocate integer variables, enumeration variables, strings and arrays of integers upon entering a subprogram was small (a few microseconds).
2. The time to allocate records (with 1, 10 and 100 integers declared in fields respectively) is substantially higher (13.9, 68.6, and 678.5 microseconds) than the time to allocate 1, 10 and 100 (0.4 microseconds) integer variables respectively. This may be due to the fact that the compiler has to align records on word boundaries.

3.4.2.2 BENCHMARK: Measure Time for Allocating Variable Amount of Storage

Variable storage allocation involves allocation of a variable amount of storage when entering a subprogram or declare block. In this test case, arrays of different dimensions bounded by variables are allocated and the size of the objects is not known at compilation time. This test is designed to determine if allocation time is dependent on size of the object. In particular, it is expected that many compilers will allocate small objects on the stack assigned to the task, and larger objects off the heap (which will typically take a much longer time). Table 6 lists the sizes and types of the objects that are allocated in these tests. In Table 6, the **size of object** column is the total number of integer elements in each array.

TABLE 6
Dynamic Allocation:Storage Allocated Is Variable
(Verdix Execution Time in microseconds)

File Name	Type Declared	Number Declared	Size of Object	Time
dd_1d1.a	1-D Dynamically Bounded Array	1	1	15.2
dd_1d10.a	1-D Dynamically Bounded Array	1	10	15.2
dd_2d1.a	2-D Dynamically Bounded Array	1	1	26.2
dd_2d10.a	2-D Dynamically Bounded Array	1	100	26.2
dd_3d1.a	3-D Dynamically Bounded Array	1	1	36.6
dd_3d10.a	3-D Dynamically Bounded Array	1	1000	36.6

- *dd_1d1.a (dd_1d10.a): Measure time to allocate and deallocate a 1-dimensional dynamically bounded array with 1 (10) integer elements upon entering a subprogram.*
  Verdix: Time for dd_1d1.a (dd_1d10.a) is 15.2 (15.2) microseconds.
- *dd_2d1.a (dd_2d10.a): Measure time to allocate and deallocate a 2-dimensional dynamically bounded array with 1 (100) integer elements upon entering a subprogram.*
  Verdix: Time for dd_2d1.a (dd_2d10.a) is 26.2 (26.2) microseconds.
- *dd_3d1.a (dd_3d10.a): Measure time to allocate and deallocate a 3-dimensional dynamically bounded array with 1 (1000) integer elements upon entering a subprogram.*
  Verdix: Time for dd_3d1.a (dd_3d10.a) is 36.6 (36.6) microseconds.

Interpretation of results:

1. The time required for dynamically bounded arrays varied approximately as a linear function of the number of dimensions. All of the ranges used in these tests were kept small in order to avoid other storage effects, such as allocating from the heap for objects above some size threshold.

3.4.2.3 Memory Allocation via the New Allocator.

In Ada, objects can be created dynamically using the new allocator. The new allocator is used to allocate various objects of different sizes and types. Allocation time of objects of type INTEGER, and ENUMERATION as well as composite type objects of various sizes are measured. This test will again show if allocation time is dependent on size (in the composite-type

object case). Also, based on these timing measurements real-time programmers can decide whether to use the new allocator for object elaboration or to declare the object as in fixed length case.

In these tests, the objects that have been allocated via the new allocator have also been freed via Unchecked_Deallocation before exiting the scope in which the object was allocated. This was done to make the tests more compatible with the tests in 3.4.2.1 and 3.4.2.2 (as in these tests the space allocated upon entering the subprogram is automatically freed upon leaving the subprogram). These tests are the same as if executed with 0 idle tasks in the system.

Table 7 lists the sizes and types of objects that are allocated in the tests with the new allocator. In Table 7, the column **size of object** for strings is STRING'LENGTH, for integer arrays **size of object** is array'length, and for records the **size of object** is specified as the number of fields in the record. The number of parameter allocated is 1 in Table 7.

TABLE 7
Dynamic Allocation with NEW Allocator
(Verdix Execution Time in Microseconds)

File Name	Type Declared	Size of Object	Time
dn_in1.a	Integer	1	59.9
dn_en1.a	Enumeration	1	59.9
dn_st1.a	String	1	59.9
dn_st10.a	String	10	59.9
dn_st100.a	String	100	59.9
dn_ar1.a	Integer Array	1	59.9
dn_ar10.a	Integer Array	10	59.9
dn_ar100.a	Integer Array	100	59.9
dn_ar1k.a	Integer Array	1000	59.9
dn_rc1.a	Record of Integer	1	75.9
dn_rc10.a	Record of Integer	10	102.9
dn_rc20.a	Record of Integer	20	205.0
dn_rc50.a	Record of Integer	50	407.0
dn_rc100.a	Record of Integer	100	746.0
dn_1d1.a	1-D Dynamically Bounded Array	1	98.9
dn_1d10.a	1-D Dynamically Bounded Array	10	98.9
dn_2d1.a	2-D Dynamically Bounded Array	1	114.0
dn_2d10.a	2-D Dynamically Bounded Array	100	114.0
dn_3d1.a	3-D Dynamically Bounded Array	1	127.9
dn_3d10.a	3-D Dynamically Bounded Array	1000	127.9

- *dn_in1.a : Measure time to allocate and deallocate 1 integer variable upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_in1.a is 59.9 microseconds.

- *dn_en1.a Measure time to allocate and deallocate 1 enumeration variable upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_en1.a is 59.9 microseconds.

- *dn_st1.a (dn_st10.a, dn_st100.a) :Measure time to allocate and deallocate strings of length 1 (10, 100) upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_st1.a (dn_st10.a, dn_st100.a) is 59.9 (59.9, 59.9)

microseconds.

- *dn_ar1.a (dn_ar10.a, dn_ar100.a, dn_ar1k.a): Measure time to allocate and deallocate an array of integer with 1 (10, 100, 1000) elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_ar1.a (dn_ar10.a, dn_ar100.a, dn_ar1k.a) is 59.9 (59.9, 59.9, 59.9) microseconds.
- *dn_rc1.a (dn_rc10.a, dn_rc20.a, dn_rc50.a, dn_rc100.a): Measure time to allocate and deallocate a record with 1 (10, 20, 50, 100) integer fields upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_rc1.a (dn_rc10.a, dn_rc20.a, dn_rc50.a, dn_rc100.a) is 75.9 (102.9, 205.0, 407.0, 746.0) microseconds.
- *dn_1d1.a (dn_1d10.a): Measure time to allocate and deallocate a 1-dimensional dynamically bounded array with 1 (10) integer elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_1d1.a (dn_1d10.a) is 98.9 (98.9) microseconds.
- *dn_2d1.a (dn_2d10.a): Measure time to allocate and deallocate a 2-dimensional dynamically bounded array with 1 (100) integer elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_2d1.a (dn_2d10.a) is 114.0 (114.0) microseconds.
- *dn_3d1.a (dn_3d10.a): Measure time to allocate and deallocate a 3-dimensional dynamically bounded array with 1 (1000) integer elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_3d1.a (dn_3d10.a) is 127.9 (127.9) microseconds.

Interpretation of results:

1. Comparing these measurements with those of the previous two sections gives application programmers an idea to the relative efficiencies of the various methods of storage allocation.
2. For the Verdix compiler, the time for dynamically bounded arrays increases linearly as the number of dimensions.
3. For the Verdix compiler, time for allocating integer records increases as the size of the record increases. In fact, the time for allocating a record of 100 integers is 746 microseconds as compared to 59.9 microseconds for allocating an integer array of size 1000. Hence, for the Verdix compiler, allocation of records takes a much longer time than allocation of arrays, integer variables, strings, and dynamically bounded arrays.

3.4.2.4 BENCHMARK: Determine the effect on time required for dynamic memory allocation when memory is continuously allocated without being freed via Unchecked_Deallocation in the scope where the memory was

allocated.

If memory is allocated in a loop via the new allocator and the memory that is allocated is not freed via Unchecked_Deallocation, then the time required for dynamic memory allocation can be affected as more space is allocated. This is due to the fact that time for dynamic allocation can depend on the state of storage management following previous allocations due to the need to recover storage and efficiently manage the available space. The tests in the previous section (3.4.2.3) had memory freed via Unchecked_Deallocation in the same scope where the memory was allocated. Hence, the timing measured in each loop was the time to allocate as well as deallocate memory. In this Section, memory that is allocated remains allocated after each timing loop is finished.
The tests and results are listed in Table 8. In Table 8, the column **size of object** for strings is STRING'LENGTH, for integer arrays **size of object** is array'length, and for records the **size of object** is specified as the number of fields in the record.

TABLE 8
NEW Allocator:No Storage Deallocation
(Verdix Execution Time in microseconds)

File Name	Type Declared	Number Declared	Size of Object	Time
dn_in1.a	Integer	1	1	51.9
dn_en1.a	Enumeration	1	1	51.9
dn_st1.a	String	1	1	50.9
dn_st10.a	String	1	10	50.9
dn_st100.a	String	1	100	51.0
dn_ar1.a	Integer Array	1	1	50.9
dn_ar10.a	Integer Array	1	10	50.9
dn_ar100.a	Integer Array	1	100	50.9
dn_ar1k.a	Integer Array	1	1000	Storage_Error
dn_rc1.a	Record of Integer	1	1	65.0
dn_rc10.a	Record of Integer	1	10	127.0
dn_rc20.a	Record of Integer	1	20	193.7
dn_rc50.a	Record of Integer	1	50	398.0
dn_rc10C	Record of Integer	1	100	736.0
dn_1d1.a	1-D Dynamically Bounded Array	1	1	87.9
dn_1d10.a	1-D Dynamically Bounded Array	1	10	88.0
dn_2d1.a	2-D Dynamically Bounded Array	1	1	103.0
dn_2d10.a	2-D Dynamically Bounded Array	1	100	103.0
dn_3d1.a	3-D Dynamically Bounded Array	1	1	115.9
dn_3d10.a	3-D Dynamically Bounded Array	1	1000	Storage_Error

- *dn_in1.a : Measure time to allocate 1 integer variable upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_in1.a is 51.9 microseconds.
- *dn_en1.a Measure time to allocate 1 enumeration variable upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_en1.a is 51.9 microseconds.
- *dn_st1.a (dn_st10.a, dn_st100.a) :Measure time to allocate strings of length 1 (10, 100) upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_st1.a (dn_st10.a, dn_st100.a) is 50.9 (50.9, 51.9) microseconds.

- *dn_ar1.a (dn_ar10.a, dn_ar100.a, dn_ar1k.a): Measure time to allocate an array of integer with 1 (10, 100, 1000) elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_ar1.a (dn_ar10.a, dn_ar100.a, dn_ar1k.a) is 50.9 (50.9, 50.9, raised STORAGE_ERROR) microseconds.
- *dn_rc1.a (dn_rc10.a, dn_rc20.a, dn_rc50.a, dn_rc100.a): Measure time to allocate a record with 1 (10, 20, 50, 100) integer fields upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_rc1.a (dn_rc10.a, dn_rc20.a, dn_rc50.a, dn_rc100.a) is 65.0 (127.0, 193.7, 398.0, 736.0) microseconds.
- *dn_1d1.a (dn_1d10.a): Measure time to allocate a 1-dimensional dynamically bounded array with 1 (10) integer elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_1d1.a (dn_1d10.a) is 87.9 (88.0) microseconds.
- *dn_2d1.a (dn_2d10.a): Measure time to allocate a 2-dimensional dynamically bounded array with 1 (100) integer elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_2d1.a (dn_2d10.a) is 103.0 (103.0) microseconds.
- *dn_3d1.a (dn_3d10.a): Measure time to allocate a 3-dimensional dynamically bounded array with 1 (1000) integer elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_3d1.a (dn_3d10.a) is 115.9 (raised STORAGE_ERROR) microseconds.

Interpretation of results:

1. In real-time embedded applications that typically run for long periods of time without allocated memory being freed, time for dynamic memory allocation can be effected as more and more space is allocated. For the Verdix compiler, time for memory allocation is not effected as more and more memory is allocated (without being freed by the application program).
2. For Verdix compiler, allocation of integer arrays of size 1000 and 3 dimensional arrays of size 1000 raised STORAGE_ERROR as the target ran out of RAM. This is due to the fact that memory allocated is not being freed and therefore, the measurements in this case exclude the time to free the memory that is being allocated. Hence, the timings listed in Table 8 are less than the timings listed in Table 7.

3.4.2.5 Memory Allocation via the New Allocator when there are active tasks in the system.

There are two scenarios:

1. Number of active tasks in the system = 5.
   Table 9 lists the results. In Table 9, the column **size of object** for STRINGs is STRING'LENGTH, for integer arrays **size of object** is array'length, and for records the **size of object** is specified as the number of fields in the record.

TABLE 9
NEW Allocator:Active Tasks = 5
(Verdix Execution Time in Microseconds)

File Name	Type Declared	Number Declared	Size of Object	Time
dn_st100.a	String	1	100	59.0
dn_ar1k.a	Integer Array	1	1000	58.9
dn_rc100.a	Record of Integer	1	100	746.9
dn_1d10.a	1-D Dynamically Bounded Array	1	10	104.0
dn_2d10.a	2-D Dynamically Bounded Array	1	100	121.0
dn_3d10.a	3-D Dynamically Bounded Array	1	1000	137.9

- *dn_st100.a :Measure time to allocate and deallocate strings of length 100 upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_st100.a is 59.0 microseconds.
- *dn_ar1k.a : Measure time to allocate and deallocate an array of integer with 1000 elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_ar1k.a is 58.9 microseconds.
- *dn_rc100.a : Measure time to allocate and deallocate a record with 100 integer fields upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_rc100.a is 746.0 microseconds.
- *dn_1d10.a : Measure time to allocate and deallocate a 1-dimensional dynamically bounded array with 10 integer elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_1d10.a is 104.0 microseconds.
- *dn_2d10.a : Measure time to allocate and deallocate a 2-dimensional dynamically bounded array with 100 integer elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_2d10.a is 121.0 microseconds.
- *dn_3d10.a : Measure time to allocate and deallocate a 3-dimensional dynamically bounded array with 1000 integer elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_3d10.a is 137.9 microseconds.

2. Number of active tasks in the system = 10. Table 10 lists the results. In Table 10, the column **size of object** for strings is STRING'LENGTH, for integer arrays **size of object** is array'length, and for records the **size of object** is specified as the number of fields in the record.

TABLE 10
NEW Allocator:Active Tasks = 10
(Verdix Execution Time in Microseconds)

File Name	Type Declared	Number Declared	Size of Object	Time
dn_st100.a	String	1	100	59.0
dn_ar1k.a	Integer Array	1	1000	59.0
dn_rc100.a	Record of Integer	1	100	746.9
dn_1d10.a	1-D Dynamically Bounded Array	1	10	102.9
dn_2d10.a	2-D Dynamically Bounded Array	1	100	121.0
dn_3d10.a	3-D Dynamically Bounded Array	1	1000	139.0

- *dn_st100.a :Measure time to allocate and deallocate strings of length 100 upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_st100.a is 59.0 microseconds.
- *dn_ar1k.a : Measure time to allocate and deallocate an array of integer with 1000 elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_ar1k.a is 59.0 microseconds.
- *dn_rc100.a : Measure time to allocate and deallocate a record with 100 integer fields upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_rc100.a is 746.9 microseconds.
- *dn_1d10.a : Measure time to allocate and deallocate a 1-dimensional dynamically bounded array with 10 integer elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_1d10.a is 102.9 microseconds.
- *dn_2d10.a : Measure time to allocate and deallocate a 2-dimensional dynamically bounded array with 100 integer elements upon entering a subprogram via the new allocator.*
  Verdix: Time for dn_2d10.a is 121.0 microseconds.
- *dn_3d10.a : Measure time to allocate and deallocate a 3-dimensional dynamically bounded array with 1000 integer elements upon entering a subprogram via the new allocator.*

Verdix: Time for dn_3d10.a is 139.0 microseconds.

Interpretation of results:

1. Tables 9 and 10 show negligible impact of existing tasks in the system on the time for memory allocation/deallocation.

3.4.3 Remarks on Memory Management

Since time and space are at a premium in real-time embedded systems, it is essential that the dynamic memory allocation and deallocation be as efficient as possible. Real-time programmers need to know the maximum time to allocate and deallocate storage for a particular Ada compiler in order to ensure that performance requirements will be met for their application.

3.5 Exceptions

Real-time embedded systems should be able to handle unexpected errors at runtime. Unexpected errors could have disastrous consequences if not handled properly. Many real-time systems operate for long periods of time in stand alone mode and there is a need for efficient and extensive error-handling for such systems. If exceptions raised during runtime are not detected, the entire situation may have to be reconstructed in real-time to determine the cause of the error. It may not always be possible to reconstruct the exact situation in real-time. The Ada exception handling mechanism provides a means by which errors can be detected and handled without catastrophic results.

3.5.1 Exception Handling Mechanism

The exception management function of the Ada runtime system is invoked when an exception is to be raised. It searches for the matching handler in the current frame. If a handler is found, control is transferred to that handler. Otherwise, the exception is propagated by raising the exception at

the point where the current frame was invoked. This is done after the exception management function simulates the "orderly return" of the frame that is thus completed.

If no handler is found for a exception that has been raised, then the exception management function invokes the task termination function of the runtime system to terminate the task or the main program in which the exception was raised. The exception management function is also responsible for raising exceptions that occur

- during a rendezvous. This exception is also raised in the rendezvous partner task.
- during task elaboration.

3.5.2 Exception Handling Tests

Four types of exception handling routines are interesting since they represent different ways in which exceptions are raised: NUMERIC_ERROR, CONSTRAINT_ERROR, TASKING_ERROR, and user-defined exceptions. The NUMERIC_ERROR exception is first discovered by the hardware and the exception is propagated back to the runtime system by an interrupt signal from the hardware. CONSTRAINT_ERROR is raised by the Ada runtime system. TASKING_ERROR is raised during task elaboration, task activation, or certain conditions of conditional entry calls (it is covered later on in Section 3.5.2.4). The user-defined exception is raised by the programmer. Except for the user-defined exception, the method of raising the exceptions can be done by forcing the relevant abnormal state in the code and by using the raise statement.

3.5.2.1 BENCHMARK: Measure a) timing overhead due to exceptions and b) exception response time when exception is handled in the block statement

e00001.a (*e00001_1.a*, *e00001_2.a*) These benchmarks measure two aspects related to exceptions when there are 0 (5, 10) idle tasks in the system:

- Measure the overhead associated with a code sequence that has an exception handler associated with it, yet no exception is raised during the execution of that code. Since exceptions are used to indicate "exceptional situations", exception handlers should gave minimum overhead during normal program execution.

- Measure exception response time for a) user-defined exception and b) pre-defined exceptions NUMERIC_ERROR, and CONSTRAINT_ERROR raised both by the raise statement as well as due to abnormal situations in the application code. Exception Response time is defined as the time when a exception is raised to the time the execution handler starts executing.

Table 11 below gives the results for exception handling times for exceptions raised and handled in a block for the Verdix compiler. In this table, the word explicit has been used for exceptions raised via the **raise** statement, and implicit is used for abnormal conditions in the code.

TABLE 11
Exception Raised and Handled in Block
(Verdix Execution Time in Microseconds)

File Name	Exception not raised	User defined explicit	Constraint _error explicit	Contraint _error implicit	Numeric _error explicit	Numeric _error implicit
e00001.a	0.2	303.3	587.5	581.8	587.8	603.9
e00001_1.a	0.2	303.8	582.7	586.4	588.7	604.9
e00001_2.a	0.2	303.8	582.7	586.4	588.7	604.9

Interpretation of results:

1. It is desirable to have minimal overhead associated with entering and exiting an exception handler's scope. If timing overhead due to exceptions is high, then time-critical applications will have unnecessary overhead even though no exception has been raised. For the Verdix compiler, the overhead associated with the code sequence (that has an exception handler associated with it, yet no exception is raised during the execution of that code) is negligible.
2. For the user-define exception, exception handling times are much less than exception handling times for other exceptions.
3. As expected, times for handling NUMERIC_ERROR (implicitly raised) is higher than exception handling times for other exceptions. NUMERIC_ERROR is raised by the underlying computing resource through the interrupt mechanism. The interrupt management function of the runtime system passes those interrupts that are traps (corresponding to predefined Ada exceptions) to the exception management function. Real-time embedded systems need fast NUMERIC_ERROR handling times to meet timing constraints.

4. Additional tasks in the system have a very slight effect on exception handling times.

3.5.2.2 BENCHMARK: Measure Exception handling time when exception is raised and propagated one level above to be handled.
User-defined, and pre-defined (CONSTRAINT_ERROR, NUMERIC_ERROR) exceptions are raised via the raise statement as well as abnormal situations in code. Depending on the exception-handling mechanism that the compiler designer chooses, exceptions may involve deallocation of stack space during exception propagation. When an exception propagates to outer scopes the system requires termination actions for each inner scope, which will involve deallocating any local space for tasks, access collections, arrays, and other dynamic structures.

- *e00002.a: No idle tasks exist in the system when the exceptions are raised.*
- *e00002_1.a: 5 idle tasks exist in the system when the exceptions are raised.*
- *e00002_2.a: 10 idle tasks exist in the system when the exceptions are raised.*

Table 12 below gives the results (for Verdix compiler) for exception handling times for exceptions raised and handled one level above: In this table, the word explicit has been used for exceptions raised via the **raise** statement, and implicit is used for abnormal conditions in the code.

TABLE 12
Exception Raised and Handled One Level Above
(Execution Time in Microseconds)

File Name	User defined explicit	Constraint _error explicit	Contraint _error implicit	Numeric _error explicit	Numeric _error implicit
e00002.a	576.2	852.2	842.5	851.6	841.8
e00002_1.a	576.2	852.2	842.5	851.6	841.8
e00002_2.a	576.2	852.2	842.5	851.6	841.8

Interpretation of results:

1. After subtracting the timings obtained in the previous section, it takes roughly about 270 more microseconds to propagate and handle the exception one level above where it is raised.

3.5.2.3 BENCHMARK: Measure Exception handling time when exception

is raised and propagated 3 and 4 levels above to be handled.
Only User-defined exceptions are raised via the raise statement. Table 13 lists the exception handling benchmarks.

TABLE 13
More Exception Handling Benchmarks
(Verdix Execution Time in Microseconds)

File Name	Benchmark Description	Time
e00003.a	User Exception handled 3 procs above	1267.8
e00003_1.a	User Exception handled 3 procs above,5 idle tasks	1267.8
e00003_2.a	User Exception handled 3 procs above,10 idle tasks	1267.8
e00004.a	User Exception Raised handled 4 procs above	1575.3
e00004_1.a	User Exception handled 4 procs above,5 idle tasks	1575.3
e00004_2.a	User Exception handled 4 procs above,10 idle tasks	1575.3

1. User-defined exception is raised and propagated 3 levels above where it handled. There are three scenarios:
   - *e00003.a: No idle tasks exist in the system when the exception is raised.*
   - *e00003_1.a: 5 idle tasks exist in the system when the exception is raised.*
   - *e00003_2.a: 10 idle tasks exist in the system when the exception is raised.*

   Verdix: For the user-defined exception, exception handling time is 1267.8 microseconds for each of the three scenarios.
2. User-defined exception is raised and propagated 4 levels above where it handled. There are three scenarios:
   - *e00004.a: No idle tasks exist in the system when the exception is raised.*
   - *e00004_1.a: 5 idle tasks exist in the system when the exception is raised.*
   - *e00004_2.a: 10 idle tasks exist in the system when the exception is raised.*

   Verdix: For the user-define exception, exception handling time is 1575.3 microseconds for all the three scenarios.

Interpretation of results:

1. This benchmark reinforces the results obtained in Section 3.5.3.2 about the extra time for each level (270 microseconds) that the exception has to be propagated.

3.5.2.4 Exception During a Rendezvous

If an exception is raised within a rendezvous, it is propagated to the task containing the accept as well as to the calling task. This is the most complex form of exception handling since the exception is handled in both the task containing the accept and the calling task. Task exception handling within a rendezvous can be quite expensive due to the overhead associated with propagating the exception to the calling environment as well as to the called task.

Table 14 lists TASKING_ERROR exception benchmarks.

TABLE 14
TASKING_ERROR Exception Benchmarks
(Verdix Execution Time in Microseconds)

File Name	Benchmark Description	Time
e00005.a	Exception Raised in rendezvous,0 idle tasks	353.0
e00005_1.a	Exception Raised in rendezvous,5 idle tasks	353.0
e00005_2.a	Exception Raised in rendezvous,10 idle tasks	353.0
e00006.a	Child task has error during elaboration,0 idle tasks	STORAGE_ERROR
e00006_1.a	Child task has error during elaboration,5 idle tasks	STORAGE_ERROR
e00006_2.a	Child task has error during elaboration,10 idle tasks	STORAGE_ERROR

3.5.2.4.1 BENCHMARK: Measure time to handle TASKING_ERROR exception in the calling task.
In this test, TASKING_ERROR is raised during a rendezvous. The same entry is timed without the exception being raised so the exception handling times can be determined.
This benchmark is executed with 3 scenarios:

- *e00005.a: No idle tasks exist in the system when the exception is raised.*

- *e00005_1.a: 5 idle tasks exist in the system when the exception is raised.*
- *e00005_2.a: 10 idle tasks exist in the system when the exception is raised.*

Verdix: TASKING_ERROR handling time is 353.0 microseconds for all the three scenarios.

Interpretation of results:

1. Propagating the exception to the calling environment as well as the called task may involve additional overhead. If task exception handling time within a rendezvous is costly when compared to exception handling time in a procedure or block, then serious consideration must be given to providing an exception handler within the accept body of the time-critical tasks.
2. The time for the Verdix compiler has to be compared to times obtained from other compiler systems.

3.5.2.4.2 BENCHMARK: Measure time to propagate and handle exception when a child task has an error during its elaboration.

Tasking_Error is raised at the place of activation of a local task (i.e. at the begin of the parent unit or on allocation of an object with a task component) if the task has an error during its elaboration.

This benchmark is executed with 3 scenarios:

- *e00006.a: No idle tasks exist in the system when the exception is raised.*
- *e00006_1.a: 5 idle tasks exist in the system when the exception is raised.*
- *e00006_2.a: 10 idle tasks exist in the system when the exception is raised.*

Verdix: The Verdix compiler raised STORAGE_ERROR during the elaboration of the child task for all the three benchmarks.

3.5.3 Remarks on Exception Handling

The problems that a compiler implementation faces in order to provide efficient exception handling mechanism for real-time embedded systems are numerous. For real-time systems, exceptions should be used only when truly unexpected situations occur and not for control flow techniques. If the exception handling mechanism is not efficient, the overhead costs involved with using exceptions will generally be too great to maintain memory and timing constraints in real-time embedded systems.

For efficient exception handling, ideally one would like that execution time be expended only when an exception is raised. There should be minimum execution time overhead in setting up and maintaining the current appropriate exception handler. Of course there are memory costs involved in maintaining the code for the exception handler.

Another factor with exception handling is the manner in which exceptions are reported and their causes. For real-time embedded system applications, a runtime system may not include TEXT_IO due to memory constraints or lack of capability to display textual messages. Hence during execution time, it may be very difficult to pinpoint the exact spot where the exception occurred.

3.6 Chapter 13 Benchmarks

Ada defines some features which allow a programmer to specify the physical representation of an entity, i.e., map the abstract program entity to physical hardware. These features are implementation-dependent: an implementation is not required to support these features. For programming real-time embedded systems, it may be necessary to use Ada LRM Chapter 13 features due to the following reasons:

- Real-time embedded systems may need to interface with physical devices and to specify the precise layout of data structures. Most Ada compilers perform memory access with logical addresses which are converted to physical addresses by the linking loader. What is needed is a way to map hardware registers, bit patterns, and addresses onto memory/source Ada objects. The solution is the implementation of bit level, address, length, record, and enumeration representation clauses.
- Unchecked_Conversion can be very useful, especially while doing communication protocols with check-sum calculations.
- Real-time applications may require explicit bit-set and bit-reset operations to manipulate external hardware devices in microprocessor systems. Bit manipulation - the direct issuance of a command to set an individual bit in memory is crucial for quick response in systems with bit-mapped I/O. Hence, the need for implementation of Chapter 13 features cannot be stressed more.

Benchmarking Chapter 13 features also depends on the characteristics listed in package SYSTEM, the hardware and its interface with the peripheral devices. The goal is to develop general purpose benchmarks that can be easily tailored for a specific implementation. Table 15 lists all the Chapter 13 benchmarks.

TABLE 15
Chapter 13 Benchmarks
(Verdix Execution Time in Microseconds)

File Name	Benchmark Description	Time
h00001.a	Boolean operations on arrays, Pragma PACK	93.3
h00001_1.a	Boolean operations on arrays, Rep Clause	91.9
h00001_2.a	Boolean operations on arrays, not packed	362.5
h00002.a	Boolean operations on array components, Pragma Pack	979.1
h00002_1.a	Boolean operations on array components, Rep Clause	968.7
h00002_2.a	Boolean operations on array components, not packed	479.4
h00003.a	Assignment,comparison on arrays of booleans, Pragma PACK	38.2
h00003_1.a	Assignment,comparison on arrays of booleans, Rep Clause	38.2
h00003_2.a	Assignment,comparison on arrays of booleans, not packed	68.2
h00004.a	Assign,compare whole records, no rep clause	61.9
h00004_1.a	Assign, compare whole records, rep clause	87.2
h00004_2.a	Assign, compare whole records, Pragma PACK	87.2
h00005.a	UNCHECKED_CONVERSION, INTEGER object to another	0.3
h00005_1.a	UNCHECKED_CONVERSION, STRING to INTEGER	1.7
h00005_2.a	UNCHECKED_CONVERSION,floating array to record	Program_Error
h00006.a	Store, extract record bit fields, no rep clause	49.1
h00006_1.a	Store, extract record bit fields, rep clause	64.1
h00006_2.a	Store, extract record bit fields, rep clause	68.7
h00008.a	Store, extract record bit fields defined by nested rep clauses using packed arrays	Not compiled
h00009.a	Change of representation from one record to another	115.9
h00009_1.a	Change of representation from one array to another	160.4
h00010.a	POS,SUCC, and PRED operations on enum type with rep clause numbered with gaps in internal coding	256.0
h00010_1.a	POS,SUCC, and PRED operations on enum type with rep clause numbered with no gaps in internal coding	256.0

3.6.1 Pragma Pack

There are a set of test problems for Pragma pack which measure both time and space utilization. Some packing methods do allocate a component so that it will span a storage unit boundary while some pack as densely as possible. The time to access a component which spans a storage unit is usually greater than when the component does not span a boundary. Although which component of a record spans a boundary is dependent on the implementation storage unit size, the computation to identify the component can be performed in an implementation independent manner using the named number SYSTEM.STORAGE_UNIT.

In addition to measuring the time to perform the test problems accessing packed objects, these test problems use the representation attribute X'SIZE to determine the actual bit size of the objects and compare this with the predetermined minimum possible bit size for the object. This shows the degree of packing the system under test performs.

3.6.1.1 BENCHMARK: This test measures time to perform standard boolean operations (XOR, NOT, OR, AND) on arrays of booleans. The tests are performed on entire arrays.
The declaration for the array is as follows:

```
subtype PACKED_16 is PACKED_BIT_ARRAY(0..15);
```

where

```
type PACKED_BIT_ARRAY is array(NATURAL range < >) of BOOLEAN;
```

The following scenarios are benchmarked:

- *h00001.a: The arrays are PACKED with the pragma 'PACK'.*
  Verdix: Time for AND, XOR, and OR operation is 93.3 microseconds. Also, ARRAY'SIZE after pragma PACK is 16 bits.
- *h00001_1.a: Representation clause is used to specify array size.*
  Verdix: Time for AND, XOR, and OR operation is 91.9 microseconds. Also, ARRAY'SIZE after representation clause is 16 bits.
- *h00001_2.a: The arrays are NOT PACKED with the pragma 'PACK'..*
  Verdix: Time for AND, XOR, and OR operation is 362.5 microseconds. Also, ARRAY'SIZE is 128 bits.

Interpretation of results:

1. Both the Pragma PACK array and representation clause array took nearly the same amount of time for the boolean operations. Also, the Verdix compiler packed the array in 16 bits for both Pragma Pack and representation clause specified array.

2. The size of the regular array (in h00001_2.a) is 128 bits (16 bytes). The time here is considerably higher than the others as in the case of h00001_2.a the boolean operations have to be performed on 16 bytes as opposed to 2 bytes in h00001.a and h00001_1.a.

3.6.1.2 BENCHMARK: This test measures time to perform standard boolean operations (XOR, NOT, OR, AND) on arrays of booleans. The tests are performed on components of arrays.
The declaration for the array is as follows:

```
subtype PACKED_16 is PACKED_BIT_ARRAY(0..15);
```

where

```
type PACKED_BIT_ARRAY is array(NATURAL range < >) of BOOLEAN;
```

The following scenarios are benchmarked:

- *h00002.a: The arrays are PACKED with the pragma 'PACK'.*
  Verdix: Time for AND, XOR, and OR operation is 979.1 microseconds. Also, ARRAY'SIZE after pragma PACK is 16 bits.
- *h00002_1.a: Representation clause is used to specify array size.*
  Verdix: Time for AND, XOR, and OR operation is 968.7 microseconds. Also, ARRAY'SIZE after representation clause is 16 bits.
- *h00002_2.a: The arrays are NOT PACKED with the pragma 'PACK'..*
  Verdix: Time for AND, XOR, and OR operation is 479.4 microseconds. Also, ARRAY'SIZE is 128 bits.

Interpretation of results:

1. Both the Pragma PACK array and representation clause array took nearly the same amount of time for the boolean operations. Also, the Verdix compiler packed the array in 16 bits for both Pragma Pack and representation clause specified array.
2. The size of the regular array (in h00002_2.a) is 128 bits (16 bytes). The time here is considerably higher than the others as in the case of h00002_2.a the boolean operations have to be performed on 16 bytes as opposed to 2 bytes in h00002.a and h00002_1.a.

3.6.1.3 BENCHMARK: This test measures time to perform assignment and comparison operations on arrays of booleans.
The declaration for the array is as follows:

```
subtype PACKED_16 is PACKED_BIT_ARRAY(0..15);
```

where

```
type PACKED_BIT_ARRAY is array(NATURAL range < >) of BOOLEAN;
```

The following scenarios are benchmarked:

- *h00003.a: The arrays are PACKED with the pragma 'PACK'.*
  Verdix: Time for assignment and comparison operations on arrays of booleans is 38.2 microseconds.
- *h00003_1.a: Representation clause is used to specify array size.*
  Verdix: Time for assignment and comparison operations on arrays of booleans is 38.2 microseconds.
- *h00003_2.a: The arrays are NOT PACKED with the pragma 'PACK'..*
  Verdix: Time for assignment and comparison operations on arrays of booleans is 68.2 microseconds.

Interpretation of results:

1. Both the Pragma PACK array and representation clause array took nearly the same amount of time for the assignment and comparison operations.
2. The time taken to execute h00003_2.a is considerably higher than the others as the operations have to be performed on 16 bytes as opposed to 2 bytes in h00003.a and h00003_1.a.

3.6.1.4 BENCHMARK: This test measures time to perform assignment and comparison operations on whole records.
The declaration of the record is as follows:

```
type MY_REC is record
    A1: INTEGER range -3 .. 3;
    A2: BOOLEAN;
    A3: INTEGER range 0..15;
    A4: INTEGER range 10 .. 100;
    A5: BOOLEAN;
end record;
```

The following scenarios are benchmarked:

- *h00004.a: Records are NOT defined by representation clause.*
  Verdix: Time for assignment and comparison operations on whole records is 61.9 microseconds. RECORD'SIZE = 128 bits (8 bytes).
- *h00004_1.a: Records are defined by representation clause.*
  Verdix: Time for assignment and comparison operations on whole records is 87.2 microseconds. RECORD'SIZE = 17 bits
- *h00004_2.a: Records are PACKED using PRAGMA PACK.*
  Verdix: Time for assignment and comparison operations on whole records is 87.2 microseconds. RECORD'SIZE = 18 bits

Interpretation of results:

1. Both the Pragma PACK array and representation clause record took nearly the same amount of time for the assignment and comparison operations (87.2 microseconds). This is considerably higher than the time for records without any Pragma PACK or representation clause (61.9). The logical explanation is that it takes more time to unpack the record in order to perform the operation.
2. The most amount of packing (17 bits) is performed by representation clause specification rather than Pragma PACK (18 bits).

3.6.2 Unchecked_Conversion

3.6.2.1 BENCHMARK: Measure the time to do an unchecked conversion of one integer object to another.

h00005.a: The time measured is for UNCHECKED_CONVERSION to move one INTEGER object to another INTEGER object. This may be nearly zero with good optimization.
Verdix: Time to do unchecked conversion is 0.3 microseconds.

Interpretation of results:

1. The time taken is nearly 0 showing good optimization by the compiler.

3.6.2.2 BENCHMARK: Measure the time for UNCHECKED_CONVERSION to move a STRING object to another INTEGER object.
h00005_1.a: This may be nearly zero with good optimization.
Verdix: Time to do unchecked conversion is 1.7 microseconds.

Interpretation of results:

1. The time taken is nearly 0 showing good optimization by the compiler.

3.6.2.3 BENCHMARK: Measure the time to do an unchecked conversion of an array of 10 floating components into a record of 10 floating components.

h00005_2.a:
Verdix: The Verdix compiler raised the exception PROGRAM_ERROR on the execution of this benchmark. **Interpretation of results:**

1. The compiler vendor has been contacted with the results.

3.6.3 Representation Clauses

3.6.3.1 BENCHMARK: Measure the time to store and extract bit fields using Boolean and Integer record components. 12 accesses, 5 stores, 1 record copy.

There are 3 scenarios:

- *h00006.a: The time to store and extract bit fields that are NOT defined by representation clauses.*
  Verdix: 49.1 microseconds
- *h00006_1.a: The time to store and extract bit fields that are defined by representation clauses.*
  Verdix: 64.1 microseconds
- *h00006_2.a: The time to store and extract bit fields that are packed by PRAGMA PACK.*
  Verdix: 68.7 microseconds

Interpretation of results:

1. Both the Pragma PACK array and representation clause record took nearly the same amount of time for operation performed in this benchmark. This is considerably higher than the time for records without any Pragma PACK or representation clause. The logical explanation is that it takes more time to unpack the record in order to perform the operation.

3.6.3.2 BENCHMARK: Measure the time to store and extract bit fields that are defined by nested representation clauses using packed arrays of Boolean and Integer record components.
h00008.a:
Verdix: The Verdix compiler could not execute this benchmark and objected to the record representation specified for the record.

Interpretation of results:

1. The compiler vendor has been contacted with the results.

3.5.3.3 BENCHMARK: Measure the time to perform a change of

representation from one record representation to another.
h00009.a:
Verdix: Time for record conversion is 115.9 microseconds. Normal record'SIZE = 128, Representation clause record size = 17

Interpretation of results:

1. The Verdix compiler took 115.9 microseconds for the record conversion. It also packed the record from 128 bits to 17 bits. This has to be compared with results obtained by running the benchmark on other Ada compilers.

3.6.3.4 BENCHMARK: Measure the time to perform a change of representation from a packed array to an unpacked array.
where

type INT is range 0..4095;
type PACKED_TYPE is array(INT range 0..315) of INT;

h00009_1.a:
Verdix: Time for array conversion is 160.4 microseconds.

Interpretation of results:

1. The Verdix compiler took 160.4 microseconds for the array conversion. This has to be compared with results obtained by running the benchmark on other Ada compilers.

3.6.3.5 BENCHMARK: Measure the time to perform POS, SUCC, and PRED operations on enumeration type with representation clause specification.
There are two cases:

- *h00010.a: Representation clause numbered with gaps in internal coding.*
  Verdix: Time for POS, SUCC and PRED operations is 256 microseconds.
- *h00010_1.a: Representation clause numbered with NO gaps in internal coding.*
  Verdix: Time for POS, SUCC and PRED operations is 256.4 microseconds.

Interpretation of results:

1. There is no effect on the execution time for representation clause with no gaps and representation clause with gaps. For enumeration type representation clauses, all compilers must generate the programmer-defined codes for enumeration types instead of its own.

3.7 Interrupt Handling

In real-time embedded systems, efficient handling of interrupts is very important. Interrupts are asynchronous events. In a real-time embedded system, interrupts are critical to the ability of the system to respond to real-time events and perform its required functions and it is essential that the system responds to the interrupt in some fixed amount of time. Interrupt handling function is responsible for handling a) software signals such as UNIX signals (for real-time UNIX operating systems), b) asynchronous hardware interrupts (real-time clock, I/O devices, and c) synchronous hardware interrupts (arithmetic exceptions). For real-time systems, interrupt latency has to be minimized. Also, interrupts should be controllable to prevent interruption of a critical section.

3.7.1 Implementation of the Interrupt Handling Mechanism

In Ada, an interrupt is identified through its association with a particular task entry. The association is effected by means of an "address" clause attached to the entry specification; the address clause, whose format is necessarily machine dependent, specifies a particular interrupt from a particular source. When an interrupt occurs a call is made to the associated entry. When the call is accepted the rendezvous is executed at a higher priority than that of any user defined task, thus ensuring that interrupt handling takes precedence over "normal" processing. The entry call may be a normal entry call or a timed entry call, depending on the kind of interrupt and on the implementation. Control information associated with the interrupt may be passed to the handling task by "in" parameters to the entry.

Because Ada treats an interrupt handling routine like any other task, compilers generate code to allocate data structures, save and store data from other tasks, and so forth, when responding to system-generated interrupts. In real-time control applications, such overhead can make or break a weapons system or flight control computer. Embedded systems must be able to suspend all other processing to react immediately--in less than a millisecond--to interrupts from an airplane navigation system sensor, for instance. The interrupt handler may enable and disable the interrupt mechanism by modifying the appropriate control registers.

3.7.2 Interrupt Handling Tests

3.7.2.1 BENCHMARK: Measure Interrupt Response Time.

Techniques for measuring interrupt response time are very difficult as hardware external to the CPU must be involved in order to generate interrupts.

This measure is totally dependent on the hardware involved, although some general criteria for measuring the interrupt response time is discussed in this report. External instrumentation (e.g., electronic equipments, real-time timers etc.) is required to accurately capture the time of interrupt occurrence. There is no difficulty in time stamping within the program the start of interrupt handler execution, but some form of external instrumentation is necessary to accurately schedule or capture the time of interrupt occurrence. In this report a general approach for measuring the interrupt response time has been described below.

Benchmark Design: The benchmark designed should be able to control the generation of interrupt signal so that the time at which the interrupt was generated can be measured. A special routine is needed that causes interrupts to be generated when specified from the benchmark program. This special routine should go through the runtime system.

The benchmark code notes the time at which the special routine is called that causes the interrupt. The interrupt handler for that interrupt records the time at which the handler was invoked and returns from the interrupt. If Tco is the overhead time for calling the clock function, and the difference in times recorded by the interrupt handler and the benchmark code is Ti, then the interrupt response time is Ti - Tco.

3.7.3 Remarks on Interrupt Handling

Because embedded systems rely heavily upon interrupts to signal the occurrences of external events, any overhead associated with the processing of the interrupts will directly affect system timing. When an interrupt is received, the runtime system has to stop the current process and issue an entry call to the interrupt handler task. In current Ada implementations, this overhead time is in hundreds of microseconds or milliseconds, whereas in non-Ada embedded systems overhead times have been in tens of microseconds or less. The treatment of interrupt priorities is not very well handled in Ada. The language does not discriminate between the hardware priorities of different interrupts. The handling of a sequence of high priority interrupts, after the first interrupt, could be blocked by the handling of lower priority interrupts. This is due to the fact that the handler of the high priority interrupt, which is required by the Ada LRM to execute a synchronization point between interrupts, is also required to have a lower priority than the low priority interrupts. Consequently, the arrival of the lower priority interrupts could block the high priority interrupt server from becoming ready to accept the second high priority interrupt after the first high priority interrupt has been handled.

3.8 Clock Function and TYPE Duration

For real-time embedded systems, the CLOCK function in the package CALENDAR is going to be used extensively. The implementation of the system clock is an important factor in the overall capabilities of the system. The CLOCK function reads the underlying timer provided by the system and returns the value associated with the timer. If the time taken to execute the CLOCK function is less than the time resolution, successive evaluations of CLOCK will return the same value. If the overhead associated with executing the CLOCK function is high, then real-time embedded systems will be hesitant to use the CLOCK function.

Most computer architectures have two kinds of hardware for timing. The counter timer chip used to drive the system clock defines the minimum granularity of time available to the system. The second level of granularity is the basic clock period which can be found in the Ada package SYSTEM (SYSTEM.TICK). Typically, some reasonable value is chosen for the size of the CLOCK period, and an interrupt is generated at this rate. The interrupt handler updates the system clock, and this represents the finest resolution

available to the CLOCK function. If the main processor is responsible for clock maintenance, as the resolution increases, so does the amount of time spent handling interrupts and maintaining the clock (this is not the case if the clock is maintained independently of the CPU).

The Ada type DURATION is not required to have the same resolution as the clock period. The smallest representable duration DURATION'SMALL must not be greater than 20 milliseconds (whenever possible a value not greater than 50 microseconds should be chosen). A real-time embedded system has timing constraints that require response within a predetermined time interval. The clock period or resolution of type DURATION must support these requirements.

Another extensive use of the CLOCK function is for the measurement of time in benchmarks. Even though more accurate timers may be present, the benchmark developer can only be certain that the system clock is present via the CLOCK function.

3.8.1 Clock Tests

Table 16 lists all the CLOCK tests.

TABLE 16
CLOCK Function Tests
(Verdix Execution Time in Microseconds)

File Name	Benchmark Description	Time
c00001.a	CLOCK function overhead	922.59
c00002.a	CLOCK resolution	0.000061

3.8.1.1 BENCHMARK: Measure CLOCK function overhead.
If the overhead associated with executing the CLOCK function is high, then real-time embedded systems will be hesitant to use the CLOCK function. The method used is essentially the same as measuring the overhead associated with a entry and exit of a do-nothing subprogram in a separate package.
c00001.a : This benchmark test measures the overhead associated with a call to and return from the CLOCK function provided in the package CALENDAR.
Verdix: CLOCK function overhead is 922.59 microseconds.

Interpretation of results:

1. The CLOCK overhead does add to the time required to make a benchmark measurement. But the dual loop benchmarking strategy can negate this effect by subtracting the control loop from the test loop.
2. For real-time applications, an overhead of 900 microseconds could be very time-expensive. Generally speaking, a CLOCK function overhead of 100 microseconds is more suitable for real-time applications. It has to be compared with the CLOCK resolution of other Ada compilers.

3.8.1.2 BENCHMARK: Measure CLOCK resolution.
If the resolution time of the CLOCK function is not high, then for real-time applications a higher resolution clock is needed.
c00002.a: This test measures the resolution time of the CLOCK function.
Verdix: CLOCK resolution is 0.000061 microseconds.

Interpretation of results:

1. The CLOCK resolution of 0.000061 microseconds is acceptable for real-time applications. Again, it has to be compared with the CLOCK resolution of other Ada compilers.

3.9 Numeric Computation

An embedded system must be able to represent real-world entities and quantities to perform related manipulations and computations. There should be support for numerical computation, units of measure (including time), and calculations and formulae from physics, chemistry etc.

3.9.1 Arithmetic for Time and Duration

For real-time embedded systems, it is necessary to dynamically compute values of type TIME and DURATION. An example of such a computation is the difference between a call to the CLOCK function and a calculated TIME value. This value may be used as a parameter in the delay statement. If the overhead involved in this computation is significant, the actual delay experienced will be longer than anticipated which could be critical for real-time systems.

3.9.1.1 BENCHMARK: Measure the overhead associated with a call to and return from the "+" and "-" functions provided in the package CALENDAR.

Times are measured for computations involving just variables and both constants and variables. The variables have predefined values. Although both "+" functions are essentially the same (only the order of parameters reversed) both are tested. This is done because a discrepancy in the time needed to complete the computation will occur if one of the functions is implemented as a call to the other. Table 17 lists the benchmarks that calculate the overhead involved in dynamic computation of values of type TIME and DURATION.

TABLE 17
TIME and DURATION Mathematics
(Verdix Execution Time in Microseconds)

File Name	Operation Performed	Time
tm1.a	Time = Var_time + Var_duration	683.20
tm2.a	Time = Var_time + Const_duration	683.20
tm3.a	Time = Var_duration + Var_time	731.99
tm4.a	Time = Const_duration + Var_time	732.60
tm5.a	Time = Var_time - Var_duration	734.99
tm6.a	Time = Var_time - Const_duration	735.20
tm7.a	Duration = Var_time - Var_time	76.50
tm8.a	Duration = Var_duration + var_duration	3.90
tm9.a	Duration = Var_duration + Const_duration	3.90
tm10.a	Duration = Const_duration + Var_duration	3.79
tm11.a	Duration = Const_duration + Const_duration	1.40
tm12.a	Duration = Var_duration - Var_duration	3.99
tm13.a	Duration = Var_duration - Const_duration	3.80
tm14.a	Duration = Const_duration - Var_duration	3.80
tm15.a	Duration = Const_duration - Const_duration	0.29

Interpretation of Results:

1. The timings of tm3.a and tm4.a (732 microseconds) are higher than tm1.a and tm2.a (683.2 microseconds). This suggests very strongly that the function "+" (Left:Duration;Right:Time) is implemented as a call to the function "+" (Left: Time; Right:Duration).
2. tm5.a and tm6.a timings are higher than tm7.a because of the time required to convert a variable or constant of type **DURATION** to type **TIME**. The time required to convert a variable of type **TIME** to type **DURATION** is small as compared to vice-versa.
3. tm8.a through tm15.a resemble timings for float addition and subtraction.

3.9.2 Mathematical Computations

TABLE 18
Numeric Computation Benchmarks
(Verdix Execution Time in Microseconds)

File Name	Operation Performed	Time
tm16.a	Float Matrix Multiplication	1567.0
tm17.a	Float Matrix Addition	1449.40
tm18.a	Factorial Calculation	133.0
tm19.a	Square root calculation	605.4

3.9.2.1 BENCHMARK: Determine time required for float matrix multiplication/addition.

- *tm16.a: A 5 by 5 matrix is multiplied by another 5 by 5 matrix.*
  Verdix: 1567.0 microseconds
- *tm17.a: A 5 by 5 matrix is added to another 5 by 5 matrix.*
  Verdix: 1449.40 microseconds.

Interpretation of Results:

1. These timings have to be compared to results from other compilers.

3.9.2.2 BENCHMARK: Determine time required for factorial and square root calculation.

- *tm18.a: Factorial of 8 is calculated.*
  Verdix: 133 microseconds
- *tm19.a: Square root of 1000 is calculated*
  Verdix: Time for SQRT calculation was 605.40 microseconds.

Interpretation of Results:

1. These timings have to be compared to results from other compilers.

3.10 Subprogram Overhead

In Ada, subprograms rank high among program units from a system structure point of view. Systems designed and implemented in Ada appear as a collection of packages and subprogram units, each of which may have multiple procedures. For real-time programmers to use good programming techniques and structured system design methodologies, it is important that subprogram call mechanism be as efficient as possible. If the subprogram overhead is high, then the compiler can generate INLINE expansion at the cost of increasing the size of the object code. However, if calls to that subprogram are made from a lot of places, then the pragma INLINE defeats the purpose due to increase in size of object code. In embedded systems where memory is at a premium using pragma INLINE may not be a practical solution. Also, a compiler implementation may not support pragma INLINE.

3.10.1 Factors Influencing Overhead of Subprogram Calls

Overhead due to subprogram calls can be high because of the following activities that take place when a subprogram is called:

- Ada has strict and elaborate rules for passing parameters to the called subprogram. These rules have to be followed when a subprogram call is made. Parameters can be passed via the stack or registers.
- Local objects are elaborated and storage for them is allocated and new exception scopes are entered for each subprogram call.

3.10.2 Subprogram Overhead Tests

Several tests were designed to provide insight to different aspects of subprogram calls. Subprogram overhead is measured when no parameters are passed in the procedure call. Then various numbers of parameters of types INTEGER and ENUMERATION are passed to determine the subprogram overhead associated with simple parameter passing. Then composite objects (arrays and records) are passed to determine if they are passed by copy or reference. Using copy to pass composite types may cause an application to incur large execution time and storage overheads. In these tests, if the time measured is constant (in other words the time does not depend on the size of the record or array passed), then parameters are passed by reference (as pass by copy times will vary with the size of the array or record passed). Finally, the formal parameters of the subprogram called are of an unconstrained composite type, thus giving the subprogram overhead involved in passing constraint information along with the parameter itself. All of the tests include passing the parameters with modes **in**, **out**, and **in out**.

All of the tests involve two different types of subprogram calls, one to a subprogram that is a part of the same package as the caller, and the other to a subprogram in a package other than the one in which the caller resides. These two sets of tests determine if there is any difference in the overhead for intra- and inter-package calls. In the case of intra-package calls, all of the tests are repeated with the addition of the INLINE pragma to determine if the INLINE pragma is supported and if it is, the amount of overhead involved in executing code generated by an in-line expansion as opposed to executing the same set of statements originally coded without a subprogram call.

The final aspect of the tests involves the use of package instantiations of generic code. All of the tests are for both inter-package and intra-package are repeated with the subprograms being part of a generic unit. These tests are designed to determine the additional overhead involved in executing generic instantiations of the code.

3.10.2.1 Intra-Package Reference Tests

In intra-package reference tests, both caller and called subprogram are part of the same package.

3.10.2.1.1 BENCHMARK: Measure the subprogram overhead involved in entering and exiting a subprogram.

Table 19 lists the types and modes of the parameters that are used in this test and also lists the results. In Table 19, the headings under the Time column: I, O, I_O have the times listed for parameters with mode **in**, **out**, and **in out**.

TABLE 19
Subprogram Overhead (Intra-Package)
(Verdix Execution Time in Microseconds)

File Name	Type of Parameter Passed	Number Passed	Size	Time		
				I	O	I_O
d_n.a		0	0	0.2		
d_i_1.a	Integer	1		10.3	10.3	9.7
d_i_10.a	Integer	10		10.3	19.5	19.5
d_i_100.a	Integer	100		89.7	150.1	190.4
d_e_1.a	Enumeration	1		9.7	9.7	9.7
d_e_10.a	Enumeration	10		10.3	20.1	20.1
d_e_100.a	Enumeration	100		90.3	150.1	190.4
d_a_1.a	Array of Integer	1	1	20.1	40.2	50.0
d_a_10.a	Array of Integer	1	10	9.7	9.7	9.7
d_a_100.a	Array of Integer	1	100	9.7	10.9	10.9
d_a_10k.a	Array of Integer	1	10000	0.6	0.6	0.6
d_r_1.a	Record of Integer	1	1	29.9	49.4	50.0
d_r_100.a	Record of Integer	1	100	0.6	0.6	0.6
d_u_a_1.a	Unconstrained array	1	1	9.7	9.7	9.7
d_u_a_100.a	Unconstrained array	1	100	9.7	9.7	9.7
d_u_a_10k.a	Unconstrained array	1	10000	9.7	9.7	9.7
d_u_r_1.a	Unconstrained record	1	1	9.7	9.7	9.7
d_u_r_100.a	Unconstrained record	1	100	9.7	9.7	9.7

Interpretation of Results:

1. It seems that the Verdix compiler is in-lining some procedures even if pragma INLINE is not specified as the timings for d_n.a and d_a_10k.a are nearly zero. The compiler vendor has been contacted with these results.

2. For integer and enumeration types, subprogram overhead for variables of mode **out** and **in out** is greater than that of mode **in**. This is because

of the additional overhead involved in copying back the parameters of mode **out** and **in out** when returning from the procedure call. Also, the overhead for passing 100 integers is higher than the overhead for passing 1 integer (due to the time required for copying the integers on the stack when the procedure call is made).

3. The timings for arrays of integer (of length 1) seemed to indicate that it is passed by copy as opposed to by reference, whereas the timings for arrays of size 10 and more seem to indicate that it is passed by reference (as pass by reference times do not vary with the length of the array passed).
4. The timings for unconstrained types seem to suggest that there is very little extra overhead in passing the constraint information in the procedure call.

3.10.2.2 Intra-Package Tests with Pragma INLINE

Many compiler implementations may not support this pragma. If pragma INLINE is supported by the compiler implementation, these tests will determine the subprogram overhead due to code generated by an in-line expansion as opposed to code that is written without a subprogram call.

3.10.2.2.1 BENCHMARK: Repeat benchmarks in Section 3.10.2.1.1 with pragma INLINE for the called procedure.

In the case of intra-package calls, all of the tests are repeated with the addition of the INLINE pragma to determine if the INLINE pragma is supported and if it is, the amount of overhead involved in executing code generated by an in-line expansion as opposed to executing the same set of statements originally coded without a subprogram call. Table 20 lists the types and modes of the parameters that are used in these tests and also lists the results in microseconds. In Table 20, the headings under the Time column: I, O, I_O have the times listed for parameters with mode in, out, and in out respectively.

TABLE 20
Subprogram Overhead (Intra-Package with Pragma Inline)
(Verdix Execution Time in Microseconds)

File Name	Type of Parameter Passed	Number Passed	Size	Time		
				I	O	I_O
i_n.a		0		0.2		
i_i_1.a	Integer	1		1.0	1.8	1.8
i_i_10.a	Integer	10		11.3	19.9	20.3
i_i_100.a	Integer	100		113.5	202.5	213.5
i_e_1.a	Enumeration	1		0.2	0.9	0.8
i_e_10.a	Enumeration	10		11.3	19.9	20.8
i_e_100.a	Enumeration	100		113.5	202.5	213.8
i_a_1.a	Array of Integer	1	1	0.5	0.4	0.4
i_a_10.a	Array of Integer	1	10	0.4	0.7	0.7
i_a_100.a	Array of Integer	1	100	0.3	0.7	0.7
i_a_10k.a	Array of Integer	1	10000	0.7	0.6	0.8
i_r_1.a	Record of Integer	1	1	0.8	1.0	1.0
i_r_100.a	Record of Integer	1	100	0.7	0.7	0.7
i_u_a_1.a	Unconstrained array	1	1	1.0	1.5	1.0
i_u_a_100.a	Unconstrained array	1	100	1.1	1.3	1.2
i_u_a_10k.a	Unconstrained array	1	10000	1.3	1.3	1.5
i_u_r_1.a	Unconstrained record	1	1	1.5	1.4	1.5
i_u_r_100.a	Unconstrained record	1	100	1.6	1.4	1.0

Interpretation of Results:

1. The overhead due to INLINE expansion of code for parameters of type integer and enumeration indicates that the overhead due to INLINE expansion is higher than the time it takes to execute the same set of statements without a procedure call. In fact, the overhead for 10 or more integer and enumeration variables is very similar to timings obtained in TABLE 19.
2. For composite and unconstrained types, the timings indicate that the overhead in executing code produced by pragma INLINE is negligible.

3.10.2.3 Inter-Package Reference Tests

In inter-package reference, the calling subprogram is in a package other than

the one in which the called subprogram resides. The motivation for inter-package tests is to compare the subprogram call overhead time between intra- and inter-package calls.

3.10.2.3.1 BENCHMARK: Repeat benchmarks in Section 3.10.2.1.1 with the called subprogram being part of another package.

Table 21 lists the types of the parameters that are used in these tests and also lists the results for the Verdix compiler. In Table 21, the headings under the Time column: I, O, I_O have the times listed for parameters with mode in, out, and in out respectively.

TABLE 21
Subprogram Overhead (Inter-Package)
(Verdix Execution Time in Microseconds)

File Name	Type of Parameter Passed	Number Passed	Size	Time		
				I	O	I_O
p_n.a		0		5.9		
p_i_1.a	Integer	1		6.8	7.3	7.5
p_i_10.a	Integer	10		14.8	19.8	24.7
p_i_100.a	Integer	100		94.7	148.6	195.1
p_e_1.a	Enumeration		1	9.2	9.1	9.3
p_e_10.a	Enumeration	10		14.8	19.9	24.7
p_e_100.a	Enumeration	100		94.8	141.6	195.1
p_a_1.a	Array of Integer	1	1	25.1	44.6	44.0
p_a_10.a	Array of Integer	1	10	6.7	7.0	6.5
p_a_100.a	Array of Integer	1	100	6.7	7.0	7.8
p_a_10k.a	Array of Integer	1	10000	7.8	7.7	7.7
p_r_1.a	Record of Integer	1	1	25.3	44.7	44.7
p_r_100.a	Record of Integer	1	100	7.0	6.9	7.0
p_u_a_1.a	Unconstrained array	1	1	7.8	8.0	7.8
p_u_a_100.a	Unconstrained array	1	100	8.0	8.0	7.9
p_u_a_10k.a	Unconstrained array	1	10000	7.9	7.8	8.0
p_u_r_1.a	Unconstrained record	1	1	6.8	9.0	8.5
p_u_r_100.a	Unconstrained record	1	100	9.1	8.6	8.7

Interpretation of Results:

1. For integer and enumeration types, subprogram overhead for variables of mode **out** and **in out** is greater than that of mode **in**. This is because of the additional overhead involved in copying back the parameters of mode **out** and **in out** when returning from the procedure call. Also, the overhead for passing 100 integers is higher than the overhead for passing 1 integer (due to the time required for copying the integers on the stack when the procedure call is made).
2. The timings for records of integer (field 1) and array of integer (of length 1) seemed to indicate that they are passed by copy as opposed to by reference, whereas the timings for records of 100 fields and arrays of size 10 and more seem to indicate that they are passed by reference (as pass by reference times do not vary with the length of the array passed).
3. The timings for unconstrained types seem to suggest that there is very little extra overhead in passing the constraint information in the procedure call. Also, unconstrained records and arrays are passed by reference.

3.10.2.4 Instantiations of Generic Code

3.10.2.4.1 BENCHMARK: In the tests for inter- and intra-package calls, the subprograms are part of generic packages that are instantiated.

These benchmarks will measure additional overhead involved in executing generic instantiations of the code. Table 22 (for intra-package) and 23 (for inter-package) list the types of the parameters that are used in these tests. In Tables 22 and 23, the headings under the Time column: I, O, I_O have the times listed for parameters with mode **in, out,** and **in out** respectively.

TABLE 22
Subprogram Overhead (Intra-Package with Generic Instantiation)
(Verdix Execution Time in Microseconds)

File Name	Type of Parameter Passed	Number Passed	Size	Time		
				I	O	I_O
g_n_c.a		0		6.0		
g_i_1_c.a	Integer	1		8.8	8.8	7.8
g_i_10_c.a	Integer	10		14.6	20.6	25.9
g_i_100_c.a	Integer	100		107.7	148.5	207.7
g_e_1_c.a	Enumeration	1		10.7	10.7	10.7
g_e_10_c.a	Enumeration	10		14.6	20.7	26.0
g_e_100_c.a	Enumeration	100		107.5	148.5	207.6
g_a_1_c.a	Array of Integer	1	1	26.4	45.7	46.0
g_a_10_c.a	Array of Integer	1	10	8.7	7.4	8.2
g_a_100_c.a	Array of Integer	1	100	8.2	7.4	8.2
g_a_10k_c.a	Array of Integer	1	10000	8.6	7.0	8.6
g_r_1_c.a	Record of Integer	1	1	26.4	45.5	46.5
g_r_100_c.a	Record of Integer	1	100	8.3	7.5	8.2

Interpretation of Results:

1. For integer and enumeration types, subprogram overhead for variables of mode **out** and **in out** is greater than that of mode **in**. This is because of the additional overhead involved in copying back the parameters of mode **out** and **in out** when returning from the procedure call. Also, the overhead for passing 100 integers is higher than the overhead for passing 1 integer (due to the time required for copying the integers on the stack when the procedure call is made).

2. The timings for integer records of field 1 and array of integer (of length 1) seemed to indicate that they are passed by copy as opposed to by reference, whereas the timings for integer records with 100 fields and arrays of size 10 and more seem to indicate that they are passed by reference (as pass by reference times do not vary with the length of the array passed).

3. There is a slight difference in the times as listed for intra-package reference without generic instantiations and with generic instantiations. On the whole, the timings seem to be compatible.

TABLE 23
Subprogram Overhead (Inter-Package with Generic Instantiation)
(Verdix Execution Time in Microseconds)

File Name	Type of Parameter Passed	Number Passed	Size	Time		
				I	O	I_O
c_n.a		0		8.9		
c_i_1.a	Integer	1		12.8	13.8	16.9
c_i_10.a	Integer	10		47.1	53.2	78.6
c_i_100.a	Integer	100		370.1	436.7	674.7
c_e_1.a	Enumeration	1		14.3	14.8	17.7
c_e_10.a	Enumeration	10		47.1	53.1	78.6
c_e_100.a	Enumeration	100		370.1	436.8	674.7
c_a_1.a	Array of Integer	1	1	50.2	69.2	89.7
c_a_10.a	Array of Integer	1	10	43.3	43.9	75.5
c_a_100.a	Array of Integer	1	100	156.8	157.4	302.6
c_a_10k.a	Array of Integer	1	10000	12646.2	12646.1	25277.4
c_r_1.a	Record of Integer	1	1	50.2	69.2	89.5
c_r_100.a	Record of Integer	1	100	156.7	157.2	302.4

Interpretation of Results:

1. For integer and enumeration types, subprogram overhead for variables of mode **out** and **in out** is greater than that of mode **in**. This is because of the additional overhead involved in copying back the parameters of mode **out** and **in out** when returning from the procedure call. Also, the overhead for passing 100 integers is higher than the overhead for passing 1 integer (due to the time required for copying the integers on the stack when the procedure call is made).

2. There is a big difference in the times as listed for inter-package reference without generic instantiations and with generic instantiations. This seems to indicate that inter-package calls with generic instantiation are extremely inefficient on the Verdix compiler as opposed to inter-package calls without generic instantiation.

3.11 Pragmas

The main purpose of pragmas is to select particular runtime features of the language or to override the compiler's default. There are certain predefined pragmas which are expected to have an impact on the execution time and space of a program. These include: SUPPRESS, CONTROLLED, SHARED, PACK, INLINE, OPTIMIZE and PRIORITY. Benchmarks for Pragma INLINE are covered under Subprogram Overhead (Section 3.10) Pragma PACK is covered under Chapter 13 benchmarks (Section 3.6). and Pragma PRIORITY is covered under Section (4.3).

These are test problems which contain the same source text where the only difference between the problems is the presence (or absence) of pragmas. Table 24 lists the Pragma benchmarks. The time column in the table below lists the improvement in execution time when the Pragma was used in the benchmark.

TABLE 24
Pragma Benchmarks
(Verdix Execution Time in Microseconds)

File Name	Benchmark Description	Time Difference
pr00001.a	Pragma SUPPRESS used for Overflow_Check, Division_Check, and Range_Check	12.8
pr00001_1.a	Pragma SUPPRESS used for Access_Check	2.2
pr00001_2.a	Pragma SUPPRESS used for Index_Check and Length_Check	94.0
pr00001_3.a	Pragma SUPPRESS used for STORAGE_CHECK	0.0
pr00001_4.a	Pragma SUPPRESS used for ELABORATION_CHECK	18.4
pr00001_5.a	Pragma SUPPRESS used for INDEX_CHECK	794.0
pr00002.a	Pragma CONTROLLED used for access type	STORAGE_ERROR
pr00003.a	Pragma SHARED	shared integer updated
pr00003_1.a	Pragma SHARED during rendezvous	shared integer updated

3.11.1 Pragma SUPPRESS

The benchmarks for pragma SUPPRESS determine the improvement in execution time when pragmas SUPPRESS is used. Pragma SUPPRESS causes the compiler to omit the corresponding exception checking (RANGE_CHECK, STORAGE_CHECK etc.) that occurs at runtime.

When the cost of runtime checks is unacceptable, the programmer can use the pragma Suppress to suppress them selectively. This can be dangerous because if a language rule is violated when the corresponding runtime checks have been suppressed, the program's behavior will become unpredictable. Nevertheless, it happens often in practice that a program with runtime checks is too large to fit into a limited memory or is too slow to meet a time constraint.

Issues relevant to the decision of whether to suppress these runtime checks include the execution overhead of performing them, the associated code size overhead, and the additional application level code needed to perform the same level of error detection if they are turned off.

3.11.1.1 BENCHMARK: Determine improvement in execution speed when pragma SUPPRESS is used for the following checks:

- *pr00001.a: Pragma SUPPRESS is used for these checks: Overflow_Check, Division_Check, Range_Check.*
  Verdix: The execution time of this test had an improvement of 12.8 microseconds when the above mentioned checks are used with pragma SUPPRESS.

  Interpretation of Results:

  1. The results imply that Pragma SUPPRESS is implemented for Overflow_Check, Division_Check, and Range_Check.

- *pr00001_1.a: Pragma SUPPRESS is used for these checks: Access_Check*
  Verdix: The execution time of this test had an improvement of 2.2 microseconds when pragma Suppress (Access_check) is used.

  Interpretation of Results:

  1. Since the improvement in execution time is negligible, it is possible that Pragma SUPPRESS for Access_Check has no effect.

- *pr00001_2.a: Pragma SUPPRESS is used for these checks: Index_check, Length_check.*
  Verdix: The execution time of this test was better WITHOUT the pragma specified. The speed without the pragma was 94 microseconds better than with the pragma specified.

Interpretation of Results:

1. The results imply that Pragma SUPPRESS besides NOT being implemented for Index_Check and Length_Check, causes a degradation in execution time when specified. The compiler vendor has been contacted with the results.

- *pr00001_3.a: Pragma SUPPRESS is used for these checks: Storage_Check.*
  Verdix: The execution time of this test had no improvement when pragma SUPPRESS (STORAGE_CHECK) is used.

  Interpretation of Results:

  1. Since the improvement in execution time is negligible, it is obvious that Pragma SUPPRESS for Storage_Check is not implemented.

- *pr00001_4.a: Pragma SUPPRESS is used for these checks: Elaboration_Check.*
  Verdix: The execution time of this test had an improvement of 18.4 microseconds when pragma Suppress (Elaboration_Check) is used.

  Interpretation of Results:

  1. The results imply that Pragma SUPPRESS is implemented for Elaboration_Check.

- *pr00001_5.a: Pragma SUPPRESS is used for these checks: Index_Check.*
  Verdix: The execution time of this test was better WITHOUT the pragma specified. The execution speed without the pragma was 794 microseconds better than with the pragma specified.

  Interpretation of Results:

  1. The results imply that Pragma SUPPRESS besides NOT being implemented for Index_Check, causes a degradation in execution time when specified. The compiler vendor has been contacted with the results.

3.11.2 Pragma CONTROLLED

Pragma CONTROLLED may be used to request that deallocation of heap objects take place only on leaving the scope of that object.

3.11.2.1 BENCHMARK: Determine if pragma CONTROLLED has any affect for a access type object.
pr00002.a: This test will allocate memory and overwrite the only access objects to the allocated memory. Pragma CONTROLLED is specified for the access object. Even if there is no garbage collection performed on the fly, STORAGE_ERROR should be raised if Pragma CONTROLLED is implemented.
Verdix: The exception STORAGE_ERROR was raised upon execution of this benchmark.

Interpretation of Results:

1. Since the Verdix compiler by default does not deallocate space upon leaving the scope of an access type object, two conclusions can be drawn:

   - Either Pragma CONTROLLED does defer deallocation of space. or
   - It is not implemented.

   Looking at the Verdix documentation it was determined that Pragma CONTROLLED is not implemented.

3.11.3 Pragma SHARED

With multiple tasks executing, there may be an instance where the same nonlocal variable must be accessed. Pragma SHARED is the mechanism that designates that a variable is shared by two or more tasks. Pragma SHARED directs the RTE to perform updates of the shared variable copies each time they are updated, but the overhead may be significant. Pragma SHARED is applied only to scalar and access type variables. Local copies of the shared variables are made identical at synchronization points, such as at the start and at the completion of the rendezvous.

3.11.3.1 BENCHMARK: Determine the overhead due to Pragma SHARED when two tasks access a shared integer variable.
pr00003.a: In this test, the main program updates a shared integer variable. This integer variable is also updated by another task. The overhead involved in updating a shared integer variable is compared to the overhead involved in updating an integer variable that is not shared.
Verdix: The overhead involved was negligible.

Interpretation of Results:

1. From the results obtained and also from looking at the compiler documentation, it was determined that the Verdix compiler does not implement pragma SHARED.

3.11.3.2 BENCHMARK: Determine the overhead in rendezvous time when a shared variable is updated during the rendezvous.
pr00003_1.a: In this test, the main program updates a shared integer variable during a rendezvous. The overhead involved in updating a shared integer variable during a rendezvous is compared to the overhead involved in updating an integer variable (that is not shared) during a rendezvous.
Verdix: The overhead involved was negligible.

Interpretation of Results:

1. From the results obtained and also from looking at the compiler documentation, it was determined that the Verdix compiler does not implement pragma SHARED.

3.11.4 Pragma PACK

Pragma PACK is covered under Chapter 13 benchmarks (Section 3.6).

3.11.5 Pragma INLINE

Benchmarks for Pragma INLINE are covered under Subprogram Overhead (Section 3.10)

3.11.6 Pragma PRIORITY

Benchmarks for Pragma PRIORITY are covered under Section 4.1.

3.12 Input/Output

Embedded systems depend heavily on real-time input and output. An Ada embedded system must have potential access to I/O ports, to control, status and data registers (for a memory mapped scheme), to direct memory access controllers, and to a mechanism for enabling and disabling interrupts. Real-time I/O is subject to strict timing requirements and can be either synchronous or asynchronous. To handle I/O for a specialized device, a special interface is needed. This interface provides the attributes found in device drivers and interrupt handlers. For example, a real-time application needs to enable, disable, and handle device interrupts; it may need to send control signals to and request status from a device; and it has to move data to and from the data registers and I/O memory of a device. Low-level asynchronous I/O operations to and from hardware devices tend to be interrupt driven. As such a real-time programmer needs low-level I/O support and the ability to handle hardware interrupts in software efficiently. Package LOW_LEVEL_IO provides the control primitives (SEND_CONTROL and RECEIVE_CONTROL) for I/O operations on a physical device. The details of this implementation is implementation dependent. For handling hardware interrupts, the Ada address clause is used to bind a particular Ada task entry to a hardware interrupt. Low level device interfaces are machine dependent and programs including them are therefore not portable. Facilities are needed for manipulating device registers, whether on architectures which use memory mapped I/O or on architectures which use specialized I/O instructions. In the case of memory-mapped I/O Ada provides representation clauses for addressing and accessing registers. For architectures with specialized I/O instructions an Ada implementation can use code inserts or the package LOW_LEVEL_IO. Low level asynchronous I/O operations to and from hardware devices tend to be interrupt driven.

3.12.1 TEXT_IO

These benchmarks deal with TEXT_IO. Table 25 lists the I/O benchmarks.

TABLE 25
Input/Output Benchmarks
(Verdix Execution Time in Milliseconds)

File Name	Benchmark Description	Time
io00001.a	Create output file and copy characters	3111.0
io00002.a	Create output file, copy data using ENUMERATION_IO	2548.0
io00003.a	Create output file, copy data using INTEGER_IO	2509.0
io00004.a	Create output file, copy data using FLOAT_IO	2563.0
io00005.a	Create output file, copy data using FIXED_IO	2549.0

ENUMERATION_IO, INTEGER_IO, FLOAT_IO, and FIXED_IO. The tests are designed to open data file for reading and copying the data to another file. Time is measured to achieve the above for each type of IO mentioned above.

io00001.a: TEXT_IO
Verdix: The Verdix compiler took 3111 milliseconds to create an output file and then copy the lines in the input file to the output file.

io00002.a: ENUMERATION_IO
Verdix: The Verdix compiler took 2548 milliseconds to create an output file and then copy the enumeration values from the input file to the output file.

io00003.a: INTEGER_IO
Verdix: The Verdix compiler took 2509 milliseconds to create an output file and then copy the integer values from the input file to the output file.

io00004.a: FLOAT_IO
Verdix: The Verdix compiler took 2563 milliseconds to create an output file and then copy the float values from the input file to the output file.

io00005.a: FIXED_IO
Verdix: The Verdix compiler took 2549 milliseconds to create an output file and then copy the fixed type values from the input file to the output file

Interpretation of Results:

1. The timing for various types of I/O varies from 2509 milliseconds to 3111 milliseconds. These timings seem expensive compared to at most a couple of milliseconds timings for other operations. For real-time systems, these results seem to indicate that TEXT_IO, ENUMERATION_IO, INTEGER_IO, FLOAT_IO, and FIXED_IO should only be performed in non-time critical regions of code.

Chapter 4: Runtime Implementation Benchmarks

The Ada Language Reference Manual (LRM) has a lot of implementation dependent features that are of concern to real-time programmers. A list of the implementation dependent features is compiled in a document published by the Ada Runtime Environment Working Group (ARTEWG) [4]. The large variance in implementation options for a feature affect application program behavior and efficiency. This is a clear signal that simply adopting the language as defined in the LRM is not enough for real-time embedded systems. The implementation approach of various Ada language features and the runtime system has to be benchmarked to assess an Ada compiler's suitability for a real-time embedded application.

This chapter deals with benchmarks that determine Ada runtime implementation dependencies. A primary source of input is the ARTEWG [4] document. The areas that have been covered include:

- Tasking
- Scheduling and Delay Statement
- Memory Management
- Exceptions
- Interrupt Handling
- Asynchronous I/O

4.1 Tasking

4.1.1 Tasking Implementation Dependencies

Table 26 lists the benchmarks that determine tasking implementation dependencies.

TABLE 26
Tasking Implementation Benchmarks
(Results for Verdix Compiler)

File Name	Benchmark Description	Results
rt_t001.a	Is task space deallocated on return from procedure on task termination	No
rt_t002.a	Is task space deallocated upon task termination when access type is declared in library unit	No
rt_t003.a	Determine order of elaboration when several tasks are activated	See 4.1.1.3
rt_t004.a	Can a task continue execution after its activation but prior to completion of activation of tasks declared in the same declarative part	Yes
rt_t005.a	If allocation of task raises STORAGE_ERROR when is exception raised	Task Creation
rt_t006.a	What happens to tasks declared in a library package when main task terminates	Do not terminate
rt_t007.a	Print default attribute STORAGE_SIZE and SIZE for tasks objects	10240 bytes 32 bytes
rt_t008.a	Order of evaluation of tasks in abort statement	See 4.1.1.8
rt_t009.a	Task aborted while updating a variable	See 4.1.1.9

4.1.1.1 BENCHMARK: Determine if task space is deallocated on return from a procedure when a task that has been allocated via the new operator in that procedure terminates.
rt_t001.a: In this benchmark, the main program program calls a procedure. Inside that procedure are declared a task type and an access type for that task type. A task is allocated (via new) in that procedure. Upon return from the procedure, the task space should be deallocated as the task type and access type are not visible outside the procedure.
Verdix: Exception Storage_Error was raised upon execution of the benchmark.

Interpretation of results:

1. Since exception Storage_Error was raised upon execution of the benchmark, this implies that task space is not deallocated upon return from the procedure.

2. In many real-time embedded systems where space is at a premium it may be desirable that task space be deallocated when that task terminates.

4.1.1.2 BENCHMARK: Determine if tasks that are allocated dynamically by the execution of an new allocator do not have their space reclaimed upon termination when access type is declared in a library unit or outermost scope.

It might be impossible for the runtime system to deallocate the task storage space after termination. This is because the access value might have been copied and an object might still be referencing the terminated task's task control block.

rt_t002.a: In this benchmark, a task is allocated (via new) whose access type is declared in a library unit.

Verdix: Exception Storage_Error was raised upon execution of the benchmark.

Interpretation of results:

1. The raising of the exception STORAGE_ERROR indicates that task space is not deallocated upon termination of a task when access type is declared in the outermost scope or in library unit.

4.1.1.3 BENCHMARK: Determine the order of elaboration when several tasks that are declared in the same declarative region are activated in parallel.

When several tasks are activated in parallel, the order of their elaboration may affect program execution.

rt_t003.a: In this benchmark, 5 tasks are declared in the main program. The task declarations are in the order TASK1, TASK2, TASK3, TASK4, TASK5 whereas the task bodies are declared in the order TASK5, TASK2, TASK1, TASK4, TASK3. Each task prints its name as soon as it is activated. The main task prints the word "master" when it is activated.

Verdix: The output of the benchmark was TASK5, TASK4, TASK3, TASK2, TASK1 and master (this was always printed in several runs).

Interpretation of results:

1. The results for the Verdix compiler indicate that the task that is declared last (NOT the task body) is elaborated and activated first, and the task elaboration order proceeds to activate the task declared before the last declared task and so on. The task declared first is activated the last and the main program starts executing after all the tasks declared in the main program's declarative part start executing.

4.1.1.4 BENCHMARK: Can a task, following its activation but prior to the completion of activation of tasks declared in the same declarative part, continue execution.

The activation of tasks proceeds in parallel. Correct execution of a program may depend on a task continuing execution after its activation is completed but before all other tasks activated in parallel have completed their respective activations.

rt_t004.a: In this benchmark, two tasks are declared in a block statement. Using Benchmark 4.1.1.3 one can determine the order of elaboration of tasks. For this benchmark to run correctly, the declaration of task REQUESTOR has to be placed so that it is activated first. Task REQUESTOR makes a conditional entry call to an entry in task SERVER. If the entry call is accepted, the value of a boolean variable is changed to TRUE else it is changed to FALSE. The entry call will only be accepted if SERVER has been activated. If the boolean variable is FALSE, this implies that a task, following its activation but prior to the completion of activation of tasks declared in the same declarative part, can continue execution.

Verdix: The Verdix compiler returned the value FALSE for the boolean variable.

Interpretation of results:

1. Since the value of the boolean variable is FALSE for the Verdix compiler, a task following its activation but prior to the completion of activation of tasks declared in the same declarative part, continues execution.

4.1.1.5 BENCHMARK: If the allocation of a task object raises the exception STORAGE_ERROR, when is the exception raised?

The LRM does not define when STORAGE_ERROR must be raised should a task object exceed the storage allocation of its creator or master. The exception must be no later than task activation: however an implementation may choose to raise it earlier.

rt_t005.a: In this benchmark, a task (A) whose STORAGE_SIZE attribute is specified to be 10 megabytes is declared in the main program.in the main program. During the elaboration of task A, STORAGE_ERROR will be raised. If STORAGE_ERROR is raised at activation of task A, then task A prints the message "STORAGE_ERROR RAISED AT TASK ACTIVATION". If STORAGE_ERROR is raised at creation of task A, then main task prints the message "STORAGE_ERROR RAISED AT TASK CREATION".

Verdix: Execution of this benchmark prints the message "STORAGE_ERROR RAISED AT TASK CREATION".

Interpretation of results:

1. The Verdix compiler raises STORAGE_ERROR at the time of task creation, rather than at the time of task activation.

4.1.1.6 BENCHMARK: What happens to tasks declared in a library package when the main program terminates?
For some real-time embedded applications, it is desirable that such tasks do not terminate. System designers need to know this information.
rt_t006.a: In this benchmark, the library task waits for the main program to terminate. If the termination of the main program causes the termination of the library task, the message "LIBRARY TASK NOT TERMINATED WHEN THE MAIN TASK TERMINATES" is not printed (the reason being that as soon as the main task terminates, tasks declared in the library package are basically terminated without any chance to write something out on the terminal).
Verdix: The message "LIBRARY TASK NOT TERMINATED WHEN THE MAIN TASK TERMINATES" is printed.

Interpretation of results:

1. For the Verdix compiler, tasks declared in library packages do not terminate when the main task terminates.

4.1.1.7 BENCHMARK: The attributes SIZE and STORAGE_SIZE provide information about storage assignments for task objects and types. These attributes can also be used to specify an exact size (amount of storage) to be associated with a task type. For some real-time applications, it is important to know how much storage a task object is allocated.
rt_t007.a: This test prints the SIZE and default STORAGE_SIZE allocated for tasks. Also, the maximum number of tasks allowed for an implementation are calculated.
Verdix: The Verdix compiler allocates 10240 storage units for the attribute STORAGE_SIZE and 32 storage units for attribute SIZE. The maximum number of allowable tasks in the system being used is 100.

Interpretation of results:

1. The default STORAGE_SIZE for tasks can be changed using the representation clause "for TASK_TYPE'STORAGE_SIZE use ...". The maximum number of allowable tasks in the system depends on the total amount of memory available to the system divided by the STORAGE_SIZE needed for a task.

4.1.1.8 BENCHMARK: Determine order of evaluation of tasks named in an abort statement.

Abort statement provides a convenient way to terminate a task hierarchy. When a task T1 aborts a task T2, the result T2'COMPLETED is true when evaluated by T1. Other tasks may not immediately detect that T2'COMPLETED is true. In real-time embedded systems, tasks may have to be aborted in a certain sequence. The semantics of the abort statement do not guarantee immediate completion of the named task. Completion must happen no later than when the task reaches a synchronization point.

rt_t008.a: There are 4 tasks in this benchmark, TASK1 has rendezvous with TASK2, and TASK3 has rendezvous with TASK4. In the accept statement of TASK4 and TASK2, a delay statement is executed. The abort statement is executed in the order TASK4 and then TASK2. If TASK4 is aborted first, then TASK3 raises TASKING_ERROR first (prints "TASK3 aborted"). If TASK2 is aborted before TASK4, then TASK1 raises TASKING_ERROR (prints "TASK1 aborted" first) before TASK3.
Verdix: The message "TASK3 aborted" is printed first.

Interpretation of results:

1. The Verdix compiler aborts the tasks in the order they are named in the abort statement.

4.1.1.9 BENCHMARK: What are the results if a task is aborted while updating a variable ?
When a task has been aborted, it may become completed at any point from the time the abort statement is executed until its next synchronization point. Depending on when an implementation actually causes the task to complete the results of an aborted task may be different. Suppose a task is updating a variable that is visible to other tasks, prior to a synchronization point. If the task is aborted just prior to the update, it may leave the variable unchanged if it becomes completed immediately, or it may update the variable and then becomes completed at the synchronization point. This could affect the results of the whole program. An implementation may defer completion of a task if it is aborted while updating a variable, and thus prevent a variable from being undefined. This may be crucial in the case of a common variable.

rt_t009.a: This benchmark determines the results if a task is aborted while updating an variable. When a task has been aborted, it may become completed at any point from the time the abort statement is executed until its next synchronization point. If a task is aborted prior to a update, it may leave the variable unchanged if it becomes completed immediately, or it may update the variable and then become completed at the synchronization point. There is 1 task declared in this program (TASK1). Main program has rendezvous with TASK1 and then after the rendezvous it aborts TASK1 which is in the process of updating X. The Main program prints the value of X. If X has been updated

then its value should be INTEGER'LAST.
Verdix: The main program prints INTEGER'FIRST.

Interpretation of results:

1. The Verdix compiler is completed before updating the variable. It does wait for the synchronization point to be reached before aborting.

4.1.2 Task Synchronization

Table 27 lists the benchmarks that determine task synchronization implementation dependencies. The results for these benchmarks are listed in their respective sections as the results cannot fit in the table column size.

TABLE 27
Rendezvous Implementation Benchmarks

File Name	Benchmark Description	Section
rt_r001.a	Algorithm used when choosing among branches of selective wait statement	4.1.2.1
rt_r002.a	Order of evaluation of guard conditions in a selective wait	4.1.2.2
rt_r003.a	Method to select from delay alternatives of the same delay in selective wait	4.1.2.3
rt_r004.a	Determine when expressions of an open delay alternative or entry family index in an open accept alternative evaluated	4.1.2.4
rt_r005.a	Determine the priority of a task which has no explicit priority specified	4.1.3.1
rt_r006.a	Determine the priority of a rendezvous between two tasks which have no explicit priorities specified	4.1.3.2

4.1.2.1 BENCHMARK: Determine algorithm used when choosing among branches of a selective wait statement.
Fairness of select-alternative is a particular aspect of scheduling fairness. If a task reaches a selective wait and there is an entry call waiting at more than one open alternative, or if a task is waiting at a selective wait and more than one open accept or delay alternative becomes eligible for selection at the same time, an alternative is selected according to criteria that are not specified in the LRM. The implementation may make a) a random selection, b) select the entry call that arrived first, c) select the first eligible

accept alternative or d) select the task with the highest priority making the entry call. Priority is not used when selecting among branches of a selective wait. Real-tim programmers need to know this mechanism as designing an embedded system without this knowledge can lead to missed deadlines even for very low levels of processor utilization.

rt_r001.a: Determine algorithm used when choosing among branches of a selective wait statement. In this benchmark, a task SELECTED is declared inside the main program with 4 entries declared in the order e1, e2, e3, and e4 in the select statement. There are four other tasks declared which make entry calls in the order e2, e4, e1, and e3 respectively. The entry calls are made such that the entry calls are queued up when SELECTED starts processing the entry calls. The order in which the entry calls are accepted is printed out.
Verdix: The Verdix compiler printed "e1 e2 e3 e4".

Interpretation of results:

1. The Verdix compiler accepts the entry calls in the order that they are declared in the select statement. This implies that real-time embedded programmers using the Verdix compiler should place their most critical accept statements at the beginning of the select statement. If a program is designed using this knowledge, it may present portability problems if the application changes the compiler for which the program was designed initially.

4.1.2.2 BENCHMARK: Determine the order of evaluation for guard conditions in a selective wait.

rt_r002.a: The main program calls entries in a task that is declared in the main program. The accept statements have guard statements in front of them. The order in which the guards are processed is printed out.
Verdix: The Verdix compiler printed "f1 f2 f4 f5".

Interpretation of results:

1. The Verdix compiler evaluates the guard conditions in the order that they are declared in the select statement.

4.1.2.3 BENCHMARK: Determine method used to select from delay alternatives of the same delay in a selective wait.

rt_r003.a: In this test, a task is declared in the main program called DELAYS. This task has an entry E1 declared along with 3 delay alternatives of "delay 1.0" in the select statement. The delay statements print the order in which they are evaluated as there is no entry call made to the accept in task DELAYS.

Verdix: The Verdix compiler printed "d1 d2 d3".

Interpretation of results:

1. The Verdix compiler always selects the first delay alternative of the same delay in a selective wait.

4.1.2.4 BENCHMARK: When are the expressions of an open delay alternative or the entry family index in an open accept alternative evaluated.

After all the open alternatives have been determined, the expressions in the open delay alternative or the entry family index in an open alternative are evaluated in the process of selecting an open alternative. It is up to the implementation to determine whether all those expressions must be evaluated before a selection is made. The choice affects both side effects and the performance behavior of the select statement.

rt_r004.a: In this test, a delay alternative is part of a select statement. The delay alternative calculates the delay time by calling a function that changes the value of a global variable X (initialized to 0). The entry call is already queued up before the select statement is reached. If the expression in the delay is evaluated before accepting the entry call, then the value of X is 10, else it is 5.

Verdix: This benchmark did not execute on the Verdix compiler. The benchmark compiled fine, but could not execute. The program never returned. The compiler vendor has been contacted.

Interpretation of results:

1. None, as the program could not execute.

4.1.3 Tasking Priorities

Ada has two features to assist in real-time control. These are the priority mechanism and timing facilities. A process priority is essential in any real-time programming language since it allows the programmer to specify that certain processes are more urgent than others and must therefore be scheduled first. In Ada a task may be assigned a priority by use of the pragma priority in its specification. Some of the consequences of the priority concept are:

- All instances of the same task type must have equal priority.

- A task cannot change its priority (other than by entering a rendezvous with a higher priority task).
- Tasks without a defined priority can have a high or low priority depending on the implementation.
- Queue of tasks on an accept statement is strictly FIFO.
- The choice among open accept alternatives of the select statement is arbitrary.

Priority is a concept that considers time in relative terms, not absolute. A task having a higher priority than another will not execute later than the second task, but no limit will be set on the time it may have to wait before execution, apart from the fact that it will not be longer than the time the lower-priority tasks have to wait. On the other hand, for a real-time tasks it is of no concern whether its priority is higher or lower than others' provided that its execution time can be guaranteed to start (or to finish) within a certain time limit.

4.1.3.1 BENCHMARK: Determine priority of tasks (and of the main program) that have no defined priority.

Real-time programmers need to know the default priority of the main program and other tasks in order to design usable embedded systems.

rt_r005.a: This benchmark determines the default priority of a task that has no priority specified. This test has tasks declared whose priorities vary from SYSTEM.PRIORITY'LAST to SYSTEM.PRIORITY'FIRST. The implementation of PRIORITY is specified in the package SYSTEM. The number of tasks declared depend on the range of value for PRIORITY. These tasks each make entry calls to an entry GO declared in a task MAIN in the main program. After the rendezvous is complete, the highest priority task should execute. If MAIN executes before the other task after the rendezvous, then the priority of task MAIN is either greater than or equal to the other task.
Verdix: For the Verdix compiler, the implementation of PRIORITY has the range 0 .. 99. 0 is the lowest priority and 99 is the highest. During the benchmark execution, the task MAIN (after the accept call) was put in the execution queue ONLY before a task of priority 0.

Interpretation Of Results:

1. The priority of tasks that have no priority specified is 0 (from a range of 0..99) on the Verdix Compiler.

4.1.3.2 BENCHMARK: Determine priority of a rendezvous between two

tasks without explicit priorities.
Two tasks without explicit priority conduct a rendezvous. If the priority given to the rendezvous is higher than a task with an explicit priority, the Ada program may perform in a manner unpredictable by the program designer. This knowledge may be required by the designers of an embedded system to ensure required system performance.

rt_r006.a: This benchmark determines the default priority of a rendezvous between two tasks with undefined priorities. This test has tasks declared whose priorities vary from SYSTEM.PRIORITY'LAST to SYSTEM.PRIORITY'FIRST. The implementation of PRIORITY is specified in the package SYSTEM. The number of tasks declared depend on the range of value for PRIORITY. There are two tasks declared MAIN and MAIN1 without explicit priorities. These tasks execute a rendezvous with each other and suspend themselves in the rendezvous via a delay. When the delay expires, all the other tasks whose priorities are higher than the rendezvous priority should execute first before the rendezvous is completed. If rendezvous is completed before other tasks which have not executed, then the priority of the rendezvous is either greater than or equal to highest priority task yet to execute.
Verdix: For the Verdix compiler, the implementation of PRIORITY has the range 0 .. 99. The rendezvous completed ONLY before a task of priority 0 was left to execute.

Interpretation Of Results:

1. The priority of an rendezvous between tasks whose priorities are not explicitly defined is 0.

4.1.3.3 BENCHMARK: Determine if a low priority task activation could result in a very long suspension of a high priority task.
A task spawning another task is the parent of that task; the parent task is delayed while the child task is activated. Any problems in activation are relayed back to the parent, who assumes responsibility for taking corrective action. During the execution of an Ada program, a low priority task spawns a task. While the activation of this spawned task is occurring, a high priority task becomes ready to execute and it remains suspended until the completion of the low priority task activation. In an embedded system, the suspension of a high priority task could prevent the response to a time critical event resulting in disastrous consequences. **This could not be benchmarked as more than one processor is needed to execute this benchmark. This processor is needed in order to have a high priority task become ready to execute while a low priority task spawns a task. This can not be accomplished with a single processor.**

4.2 Scheduling and Delay Statement

Task scheduling is an important consideration for a multitasking application. Real-time embedded systems contain jobs with hard deadlines for their execution. Failure to meet a deadline reduces the value of the job's execution possibly to the extent of jeopardizing the system's mission. It is the responsibility of the runtime system's scheduling mechanism to guarantee that the most important deadlines are met while also meeting as many of the less important deadlines as possible. Also, the scheduler has the responsibility to allow execution among tasks.

In MacLaren [8], real-time applications are classified by their inherent scheduling complexity as follows:

- Purely cyclic (periodic) applications: Schedules are rigid and invariant since no asynchronous events will occur. A cyclic executive can be used to implement such applications. With a cyclic executive, the application programmer controls the scheduling.
- Cyclic applications with some asynchronous events and possible variations in computing loads. (Ada multitasking approach can be used to solve these problems)
- Event driven applications that contain little or no periodic processing. These kinds of applications can also be used using Ada multitasking.

The LRM imposes no restrictions on scheduling of tasks, except that a task of higher priority that is ready to execute will not be kept waiting while a task of lower priority executes.

For scheduling tasks at a particular time, the **delay** statement can be used in conjunction with the predefined CALENDAR package. The time given in the delay statement is expressed in fractions of seconds and is a fixed point type defined in the predefined package CALENDAR. The delta of this fixed point type DURATION is implementation dependent and is defined by the value of the attribute SMALL. The precision of the timing depends on the implementation of the package CALENDAR and on the granularity of the underlying scheduler.

Some of the points to note about the delay statement are:

- The semantics for the delay statement, however, provides only that the delay specified is a minimum amount of the delay time. On average, a delay is likely to be longer than specified because of the time it takes for the runtime system to recognize the expiration of the delay, reschedule and resume the task. Furthermore, delay can only be used for coarse timing. Fine resolution timing requires an external clock and interrupt.

For real-time embedded systems, it is the maximum delay not the minimum delay which is of interest.

- There is the possibility of an interrupt between the time a delay is computed and the time it is requested. Hence the time at which delay expires cannot, in general, be predicted in advance.

Ada's preemptive, priority-based tasking model is inherently nondeterministic. Predictability and reliability may demand that portions of a given multitasking system be deterministic. Process execution order and repetition rate must be fixed in such systems. The basic problem is that the Ada rules are oriented towards solving the problems of task starvation rather than towards the kind of scenarios that arise in practice with deadline scheduling.

To allow execution to switch among tasks, the scheduler provided by the runtime system is entered at certain synchronization points in a program, and the scheduler decides at this point which task has to be executed. According to the LRM, a implementation is free to choose among tasks of equal priority or among tasks whose priority has not been defined. For some applications fairness within a priority level may be desirable. For other applications an unfair scheduler may be preferable.

The minimum synchronization (a implementation may choose to have more) points at which the scheduler is invoked are the beginning and end of task activations and rendezvous. The pragma priority enables real-time embedded systems programmers to specify a higher priority for more important tasks. The priority is fixed at compile time (we are assuming that pragma priority is implemented). Hence, whenever a scheduling decision has to be made, the highest priority task receives control (task priorities are discussed in Section 4.1.3).

The real-time performance of a system is highly dependent upon the performance of the scheduler, which in turn, is highly dependent upon the timing mechanisms available. Accordingly, real-time computer systems nearly always contain an interval timer, with either a fixed or programmable time interval.

Table 28 lists the benchmarks for Scheduling and delay statement dependencies.

TABLE 28
Scheduling and Delay Statement Dependencies
(Results for Verdix Compiler)

File Name	Benchmark Description	Results
dt00001.a	Determine minimum delay time	0.001 sec
dt00002.a	Determine if user tasks are pre-emptive	Yes
dt00003.a	Determine method to share processor within each priority level	See 4.2.3
dt00004.a	Does delay 0.0 cause scheduling	Yes

4.2.1 BENCHMARK: Determine the minimum delay time.

dt00001.a: This benchmark determines the actual delay time for a desired delay time specified in the delay statement. This benchmark starts by calculating the actual delay time for a minimum delay of DURATION'SMALL. The desired delay time is increased in steps and the actual delay calculated.
Verdix: The Verdix compiler had a minimum actual delay of 0.001 seconds for a desired delay of 0.000061 seconds (DURATION'SMALL). As the desired delay time was increased, the actual delay time remained the same till the desired delay time was 0.0005 seconds. The actual delay time then became 0.002 seconds. The actual delay time remained 0.002 seconds for a desired delay time of 0.001 seconds.

Interpretation of Results:

1. The results show that even if a delay time of DURATION'SMALL is specified (0.000061 seconds) the minimum actual delay for the Verdix compiler is 0.001 seconds. This is quite a big variation and real-time programmers definitely need to be aware of this when using the Verdix compiler.

4.2.2 BENCHMARK: Determine if user tasks are pre-emptive. Does a completed delay interrupt the currently executing task to allow the scheduler to elect the highest priority tasks.

dt00002.a: In this test, a task of a lower priority task (B) is executing when a high priority task (A) becomes eligible for execution. If user tasks are pre-emptive then A should interrupt B and resume execution.
Verdix: Task A interrupted task B after it became eligible for execution.

Interpretation of Results:

1. The Verdix compiler has delays that are pre-emptive.

4.2.3 BENCHMARK: Determine the method of sharing the processor within each priority to prevent starvation of any single task.

The Ada LRM does not specify the method by which a scheduler should choose among tasks of equal or unstated priority. An implementation may choose round-robin scheduling or some arbitrary method to choose among tasks of equal priority. Another implementation may choose to implement time slicing. Although overhead is required to implement time slicing, it is a good way to insure that each task within a priority will get an even chance at processing time. If time-slicing has been implemented in conjunction with pre-emptive priority scheduling, the algorithm must take into consideration the time remaining in the slice allotted a task that gets pre-empted so that it will be allowed to finish its slice when scheduling returns to that priority level.

dt00003.a: In this benchmark, 3 tasks are executing at the same priority (A, B, C). These tasks have no synchronization points so that the scheduler is not invoked by the tasks themselves. If either of the three tasks runs to completion without the other 2 being able to execute, then the scheduler is starving tasks within the same priority level.

Verdix: This benchmark was run with a) the runtime system configured without time-slicing and b) configured with time-slicing. For the case without time-slicing, the results show that task C ran to completion, then task B, and finally task A. When time-slicing was enabled then the three tasks ran in a round robin fashion.

Interpretation of Results:

1. For the Verdix compiler, if time slicing is not enabled then tasks of the same priority execute to completion unless a synchronization point is reached.
2. For the Verdix compiler, if time slicing is enabled then tasks of the same priority are served in a round robin fashion.

4.2.4 BENCHMARK: Does delay 0.0 simply return control to the calling task or causes scheduling of another task.

The delay statement suspends further execution of the task that executes the delay statement, for at least the duration specified by the value of the expression in the delay statement. There is an overhead in requesting an delay from the runtime system. This overhead may exceed the requested delay period. Unless the requested task has been aborted, the runtime system may return control directly to the task or schedule another task for execution.

dt00004.a: In this benchmark, the main program calls another task entry which has a delay 0.0 statement in the accept. The execution of the delay 0.0 should either cause scheduling action or be ignored.

Interpretation of results:

1. Delay 0.0 for the Verdix compiler causes scheduling of another task and does not simply return control to the calling task.

4.3 Memory Management

One of the biggest reliability concerns in real-time embedded systems are heap deallocation issues. If storage that is allocated is never reclaimed, then there is grave danger of the system running out of storage. Real-time embedded systems are designed to operate for indefinitely long periods and running out of storage at a critical point could be disastrous for the application. Therefore in real-time systems it may not be acceptable to allocate space and not deallocate it all. For space deallocation, some systems use garbage collection mechanism. If an implementation interrupts all processing to perform garbage collection, there is a danger that timing constraints in real-time systems may not be met. This garbage collection is either run periodically or when the amount of allocated memory reaches some threshold. Garbage collection runs on the same CPU as the application program and can take an inordinate amount of time at irregular intervals. Some real-time applications may not require garbage collection to be performed by the runtime system and make storage deallocation the responsibility of the application program.

It is important for real-time programmers to know if a particular compiler implementation:

- deallocates nothing
- supports only UNCHECKED_DEALLOCATION
- deallocates all the storage for an access type when the scope of the access type is left
- detects inaccessible storage and automatically deallocates it (garbage collection).

Table 29 lists the benchmarks memory management dependencies.

TABLE 29
Memory Management Dependencies
(Results for Verdix Compiler)

File Name	Benchmark Description	Results
m00001.a	Determine STORAGE_ERROR threshold	1.2 meg
m00002.a	Is Unchecked_Deallocation implemented	Yes
m00003.a m00003_1.a	Garbage Collection performed on fly	No
m00004.a	Garbage Collection performed on scope exit	No

4.3.1 BENCHMARK: Determine STORAGE_ERROR threshold.
This benchmark depends on the amount of memory available to the system and hence comes more under runtime implementation benchmarks rather than under micro benchmarks.

m00001.a: This test is basically concerned with determining at which point exception STORAGE_ERROR is raised. If memory is allocated in a loop via the new allocator, and the access variable that is pointing to the allocated memory remains throughout the run, then STORAGE_ERROR will be raised at some point.
Verdix: For the system that these benchmarks were run on, 340 arrays of 1000 integers was the maximum storage space allocated. At this point STORAGE_ERROR was raised.

Interpretation Of Results:

1. The size of the memory space available is approximately 1.2 megabytes. This is the size at which STORAGE_ERROR should be raised.

4.3.2 BENCHMARK: Determine if Garbage collection is performed on the fly.

m00003.a, m00003_1.a: These tests use the same loop structure as 4.3.1, but only two access variables. Each time around the loop, the contents of one access variable is shifted to the second, and the newly acquired data is assigned to the first access variable, thus implicitly freeing the storage allocated two iterations previous to the current one. If there is no garbage collection

performed on the fly, STORAGE_ERROR will be raised at the same point as in 4.3.1.
Verdix: STORAGE_ERROR is raised.

Interpretation Of Results:

1. The Verdix compiler raised STORAGE_ERROR. Hence no implicit deallocation is performed.

4.3.3 BENCHMARK: Determine if Garbage collection is performed on scope exit.

m00004.a: In this test an access type to an array of 10000 integers is declared in a procedure called from the main program. This subprogram is called repeatedly and if storage is not being automatically deallocated upon scope exit, STORAGE_ERROR will again be raised. If garbage collection is implicitly called, no STORAGE_ERROR exception will be raised.
Verdix: STORAGE_ERROR is raised.

Interpretation Of Results:

1. The Verdix compiler raised STORAGE_ERROR and hence no Garbage Collection is performed on scope exit.

4.3.4 BENCHMARK: Determine if Unchecked Deallocation is implemented.

m00002.a: The structure of this test is the same as the STORAGE_ERROR threshold test, except that UNCHECKED_DEALLOCATION is added to explicitly free storage. If STORAGE_ERROR is raised again, then UNCHECKED_DEALLOCATION is not implemented.
Verdix: No STORAGE_ERROR is raised.

Interpretation Of Results:

1. The Verdix Compiler has Unchecked_Deallocation implemented.

4.4 Exceptions

4.4.1 BENCHMARK: Does an implementation raise NUMERIC_ERROR on an intermediate operation when the larger expression can be correctly computed ?

Some compilers may not raise NUMERIC_ERROR for the above scenario.

rt_e0001.a: In this benchmark, an integer variable A whose initial value is INTEGER'LAST is involved in an expression of the form:

$$A:=(A+1)-2;$$

Determine if this raises NUMERIC_ERROR.
Verdix: The Verdix Compiler raises NUMERIC_ERROR.

Interpretation Of Results:

1. The Verdix compiler raises NUMERIC_ERROR on an intermediate operation even when the larger expression is correctly computed.

4.5 Interrupt Handling

For interrupt handling tests, hardware external to the testbed system is needed. The hardware is needed to generate interrupts. Also a logic analyzer is needed to capture the time of interrupt occurrence. Also, some of the questions that are listed below are hidden in the runtime system code and can only be obtained from the compiler implementor. The following information about interrupt handling is needed by the software designers

- Determine if an interrupt entry call is implemented as a normal Ada entry call, a timed entry call, or a conditional entry call.
- What are the implementation restrictions on these interrupt entries. Can they be called from the application code? Can they have parameters ?
- Determine if an interrupt is lost when an interrupt is being handled and another interrupt is received from the same device.
- Determine the restrictions imposed by an implementation for selection of the terminate alternative that may appear in the same select statement with an accept alternate for an interrupt entry. Selecting the terminate alternative may complete the task which contains the only accept statements which can handle the interrupt entry calls, leaving the hardware unserviced.
- Determine if an interrupt entry call invokes any scheduling decisions. An interrupt need not invoke any scheduling actions.
- Determine if accept statement executes at the priority of the hardware interrupt, and if priority is reduced once a synchronization point is reached following the completion of accept statement.

Consider an interrupt handler that is coded below:

```
task high_priority_handler is
    entry interrupt; for interrupt use at ....; -- address clause
    pragma priority(..); --high task priority;
end;

task body high_priority_handler is
begin
    loop
        accept interrupt do
          -- this code is executed at hardware (highest) priority
        end;
        -- this code is executed at task priority, allowing low
        -- low priority devices to interrupt
    end loop;
end;
```

The handler deals with high priority interrupts, and is therefore allocated a high task priority. However, it can be interrupted outside the rendezvous by a low priority interrupt and cannot guarantee to return to the accept statement in time to catch the next high priority interrupt.

- Determine if interrupt entries can be called from application code.

4.6 Asynchronous I/O

One of the benefits of Ada's tasking techniques is the ability to implement true asynchronous I/O. By using Ada tasks to drive I/O controllers, only the task that requested the I/O must wait for completion before resuming execution, while other tasks within the application program can continue execution while I/O is being processed. For example, suppose task1 has requested I/O from a device. While waiting for completion, task1 places itself in a wait state, letting task2 begin execution. Since task2 also wants I/O from the same device as task1, its rendezvous request is placed on the queue and task2 suspends, pending completion of the rendezvous. At this point task3 is free to execute.

4.6.1 BENCHMARK: Determine if true asynchronous I/O is implemented.

rt_io001.a: In the main procedure, two separate tasks are activated. Task1 is the highest priority task, and task2 is medium priority task. Task1 makes a

request from a I/O device, then task2 makes a request to the same I/O device. Both task1 and task2 should be suspended and and the main program should be executing at this point.
Verdix: The main task does not execute when tasks task1 and task2 were waiting for I/O.

Interpretation of Results:

1. The Verdix compiler does not implement true asynchronous I/O.

Chapter 5: Real-Time Paradigms

Users, system programmers, and academicians have found a number of useful paradigms for building concurrency. Real-time systems will be designed as a set of cooperating concurrent processes (Ada tasks) using the Ada tasking model. Translation of concurrency paradigms may force the creation of intermediary tasks with the risk of compromising real-time performance. There is a runtime penalty when intermediary tasks are introduced into the design. The more tasks introduced, the higher the penalty in the form of additional task control blocks, task scheduling and dispatching time, and context switching time.

Real-time paradigms can be coded in Ada using macro constructs and benchmarked. Also, a compiler implementation may recognize these paradigms and perform optimizations to implement that paradigm much more efficiently.

Table 30 lists real-time paradigms that have been benchmarked in this chapter.

5.1 Intermediary Tasks

Many real-time implementations require buffered and unsynchronized communication between tasks. Rendezvous is the mechanism used in Ada for task communication. Due to the rendezvous being a synchronous and unbuffered message passing operation, intermediary tasks are needed to uncouple the task interaction to allow tasks more independence and increase the amount of concurrency.

TABLE 30
Real-time Paradigms
(Verdix Time in Microseconds)

File Name	Benchmark Description	Time
pa00001.a	Simple producer consumer transaction with main calling consumer task	358.6
pa00001_1.a	Simple producer consumer transaction with consumer using selective wait	419.8
pa00001_2.a	Simple producer consumer transaction with producer task calling consumer task	358.6
pa00001_3.a	producer task communicates with consumer task through a bounded buffer	954.9
pa00001_4.a	producer task communicates with consumer task indirectly through a bounded buffer with a transporter between buffer and consumer	1248.8
pa00001_5.a	producer task communicates with consumer task indirectly through a bounded buffer with a transporter between buffer and producer as well as transporter between buffer and consumer	1701.1
pa00001_6.a	Producer task communicates with a consumer via relay	634.9
pa00002.a	Monitor using semaphores	Error
pa00002_1.a	Monitor using rendezvous	1801.0
pa00002_2.a	Monitor using rendezvous	Error
pa00003.a	Selection of Highest Priority client during entry call	Compile Error
pa00004.a	Abort a task and create a new one	4253.0

Various combinations of intermediary tasks are used in different task paradigms to create varying degrees of asynchronism between a producer and consumer. Intermediary tasks introduce a lot more rendezvous in a real-time system than if a producer and consumer were directly communicating with each other. The use of intermediaries also adds to the cost of executing a real-time design in Ada. The benchmarks in this section evaluate the cost of introducing intermediary tasks for various real-time tasking paradigms. The goal of these benchmarks is to give real-time programmers a feel for the cost of using such paradigms in a real-time embedded application and to avoid using such paradigms if the cost is unacceptable for a real-time system.

5.1.1 Producer-Consumer

The case where one task passes information to another task is called a producer-consumer relationship. The task that is the source of the information is called the producer and the task that is the recipient of the information is called the consumer. Each time a piece of information is passed to the consumer one rendezvous occurs.

5.1.1.1 BENCHMARK: Measure time for a simple producer-consumer type transaction when the main procedure calls a consumer task.

pa00001.a: The main task calls a consumer task. A simple integer value is the only data transferred and the consumer simply loops on the accept statement. Task/activation/termination time is not included in the timing. The time measured is the time it takes for the producer to call the entry in the consumer, the start of rendezvous with the consumer accepting the information, and the beginning of execution of the calling task. This is equivalent to two context switches: the first from the main task to the called task and the second from the called task to the main task.
Verdix: Time for a single rendezvous is 358.6 microseconds.

5.1.1.2 BENCHMARK: Measure time for a producer-consumer type transaction when the consumer uses a selective wait.

pa00001_1.a: In this test the main task calls a consumer task that consumes more than one type of item. A simple integer value is the only item transferred and the consumer simply loops on the selective accept. This test differs from the previous test in that the consumer uses a select statement to take the entry call where the select has two open alternatives. In the previous case there is no select statement. Task/activation/termination time is not included in the timing. The time measured is the time it takes for the producer to call the entry in the consumer, the start of rendezvous with the consumer accepting the information, and the beginning of execution of the calling task.
Verdix: Time for rendezvous in this case is 419.8 microseconds.

5.1.1.3 BENCHMARK: Measure time for a producer-consumer type transaction when a producer task calls a consumer task.

pa00001_2.a: This is similar to test 5.1.1.1, except that a producer task calls an entry in the consumer task, instead of the main task calling an entry in the consumer task. Both the producer and consumer task have the highest

priority possible (PRIORITY'LAST).
Verdix: Time for a single rendezvous is 358.6 microseconds.

Interpretation of Results:

1. Each time a piece of information ia passed from the producer to the consumer one rendezvous occurs. The time required for this type of interaction was measured to be 358.6 microseconds without a select to 419.8 microseconds with a select.
2. pa00001.a, pa00001_1.a, and pa00001_2.a are examples of tasks with tight coupling. Tight coupling is obtained via the Ada rendezvous without introducing any intermediary tasks. The advantage of a direct rendezvous between the producer and consumer is that no intermediary tasks are introduced with an associated runtime overhead.

5.1.2 Buffer Task

A buffer is pure server task that provides for one entry for storing of items in a buffer and another entry for providing items from the buffer. A buffer uncouples the producer from the consumer to provide more independence. Since the buffer is a task, its use adds some overhead. Both the consumer and the producer call the buffer task to obtain a piece of information.

pa00001_3.a: In this benchmark, the producer task communicates with the consumer task indirectly through a bounded buffer.
Verdix: Time taken by the consumer to receive information from the producer via the buffer task is 954.9 microseconds.

Interpretation of Results:

1. Each time a piece of information is passed from the producer to the consumer two rendezvous occur: the producer with the buffer and the consumer with the buffer. The time required for this type of interaction was measured to be 954.9 microseconds.
2. This is an example of a loose coupling between the producer and the consumer.

5.1.3 Use of a Buffer and Transporter

Many times a producer will want to communicate with a consumer via a buffer, but it is undesirable for the consumer to be a calling task. For example, the consumer may want to accept request from any number of producers and therefore would want to be a called task. This is accomplished by having a transporter task take information from the buffer and pass it on to the consumer i.e., producer-buffer-transporter-consumer. This scheme implies the use of two intermediary tasks between the producer and the consumer.

pa00001_4.a: In this benchmark, a producer task communicates with a consumer task indirectly through a bounded buffer with a transporter between the buffer and the consumer.
Verdix: Time taken by the consumer to receive information from the producer via the buffer and transporter tasks is 1248.8 microseconds.

Interpretation of Results:

1. Each time a piece of information is passed from the producer to the consumer three rendezvous occur: the producer with the buffer, the transporter with buffer, and the transporter with the consumer. The time required for this type of interaction was measured to be 1248.8 microseconds.

5.1.4 Use of a Buffer and Two Transporters

If both the producer and consumer wish to communicate via a buffer and both need to be called tasks, it is necessary to use a transporter on each side of the buffer. This results in the producer-transporter-buffer-transporter-consumer paradigm.

pa00001_5.a: In this benchmark, a producer task communicates with a consumer task indirectly through a bounded buffer with a transporter between the buffer and the producer as well as between the buffer and the consumer.
Verdix: Time taken by the consumer to receive information from the producer is 1701.1 microseconds.

Interpretation of Results:

1. Four rendezvous occur when a piece of information is passed from the consumer to the producer. These are - a transporter with the producer, the transporter with the buffer, a second transporter with the buffer, and the second transporter with the consumer. The time required for this type of interaction was 1701.1 microseconds.

5.1.5 Use of a Relay

A relay is an intermediary task that takes information from a producer and passes it on to the consumer.

pa00001_6.a: In this benchmark, a producer task communicates with a consumer via the relay.
In terms of the task communication model, this resembles the producer-buffer-transporter-consumer paradigm, but in terms of performance it should resemble the producer-buffer-consumer paradigm.
Verdix: Time taken by the consumer to receive information from the producer is 634.9 microseconds.

Interpretation of Results:

1. For each piece of information that is passed from the producer to the consumer two rendezvous occur - the producer with the relay and the relay with the consumer. The time required for this type of interaction was measured to be 634.9 microseconds.

5.2 Asynchronous Exceptions

Quick restarts of tasks are required in a number of real-time embedded systems. Ada model of concurrency does not provide an abstraction where a task may be asynchronously notified that it must change its current execution state. In the initial design of the language, FAILURE exception was intended to serve this purpose. But FAILURE exception was removed as it is difficult to implement as there are complicated interactions between FAILURE and other exception conditions, or multiple instances of FAILURE.

One way to implement asynchronous change in control is to abort the task and then replace it with a new one. Aborting a task may not be appropriate for an application because an abort can take a long time to complete or because the asynchronous change of control needed is something other than termination.

pa00004.a: This benchmark measures the time to abort a task and create a new task.
Verdix: The Verdix compiler took 4253.0 microseconds to abort an existing task and spawn a new task.

Interpretation of Results:

1. Abort and task initialization are expensive operations and a abort could take a long elapsed time to complete. The Verdix compiler takes a rather large amount of time to abort and create a new task.

5.3 Selection of Highest Priority Client

The LRM states that in a select statement if more than one accept is open and ready for a rendezvous, then any one accept can be chosen and the choice is left to the compiler implementor. In real-time embedded systems, it may be necessary to choose the highest priority waiting client.

When a set of client tasks directly request service from a server by calling a server's entry, the requests are processed by the server in the order of their arrival. But in real-time systems, it may be essential that the clients should be processed in the order that corresponds to their priority rather than their arrival.

pa00003.a: This benchmark implements a generic package that orders client requests so that they are processed by the server in a priority order. This package logically exists as an intermediary between the clients and the server.
Verdix: The Verdix compiler could not compile this program, although this program can be compiled on other systems. The compiler during the semantic analysis phase just dumped core. The compiler vendor has been contacted with the results.

Interpretation of Results:

1. The overhead to this solution is three additional rendezvous for each prioritized rendezvous. This is pretty expensive in terms of execution time for a prioritized rendezvous.

5.4 Monitor/Process Structure

A monitor is commonly used for controlling a systems resource. Such a task performs a watchdog function and would be classified as an actor task (Actor tasks are active in nature and make use of other tasks to complete their function). For example, read and write operations to a disk are usually controlled by a monitor that ensures the integrity of data on the disk. This is

also known as mutual exclusion. Monitors can be implemented to have controlled access to a shared data pool. Monitors can be implemented via semaphores, event signaling, [12] and rendezvous mechanism. The implementation via semaphores and event signaling is essentially the same.

Semaphores are an effective low-level synchronizing primitive. However, the use of semaphores in an complex application can result in disaster if an occurrence of a semaphore operation is omitted somewhere in the system or if the use of a semaphore is erroneous. A monitor replaces the need to perform operations on semaphores. Entry to a monitor by one process excludes entry by any other process. A monitor thereby ensures that if it has exclusive access to a resource, then a monitor's user has exclusive access to that resource. In [10], several approaches to implementing a monitor are discussed.

In this program, a monitor is developed in Ada. The problem is having a pool of data common to a group of processes. The data in the pool may be set by one or more processes or used by one or more processes. Any number of processes are allowed to read the pool simultaneously, but no reads are permitted during a write operation. The monitor developed is used to control the reading and writing of data to the pool.

Two implementations are considered:

- *pa00002.a* the first using semaphores;

  Verdix: The Verdix compiler at runtime just exited with status 0. This is a bug in the Verdix compiler as the program exited without even executing the first statement of the program.

- the last two (*pa00002_1.a, pa00002_2.a*) using the Ada rendezvous mechanism.

  Verdix: The time to do a single read and write operation to the pool (pa00002_1.a) was calculated to be 1801 microseconds.

  Verdix: The Verdix compiler raised TASKING_ERROR on execution of this benchmark.

Interpretation of Results:

1. The macro construct semaphore is used to implement this monitor/process structure for controlling access to a shared data pool. Based on the timings for the semaphore and the rendezvous implementations, real-time programmers can choose the best implementation mechanism.

5.5 Mailbox

In message passing, a question that arises is where messages are to be deposited. A common paradigm involves "mailboxes" (mailbox is a macro construct) which are global variables updated by processes to provide asynchronous communication. These are specially suitable for such situations as the producer/consumer scenario in which a producer produces some output which is consumed by a consumer process. The mailbox implementation of this involves a global mailbox visible to both these processes, and a send operation by the producer into this mailbox. The consumer then performs a receive operation on the mailbox to retrieve the data.

This paradigm is similar to the buffer task paradigm in Section 5.1.2 (pa00001_3.a) in the sense that each time a piece of information is passed from the producer to the consumer two rendezvous occur..

Chapter 6: Conclusions

This report has developed a series of benchmarks to test the performance of Ada compilers meant for real-time embedded systems. Finding an efficient and reliable cross-compiler for programming embedded systems (even for widely used and powerful micro-processors such as the Motorola 68000 family) is extremely difficult. The embedded programmer community is faced with trying to use incomplete and somewhat unstable systems while trying to deliver efficient and ultra-reliable code. This benchmarking effort concentrated on developing benchmarks that measure

- the runtime performance of Ada code on a bare target system,
- the runtime system implementations of various features of a particular Ada compiler system,
- and the performance of commonly used real-time paradigms.

It is hoped that the results of this benchmarking effort will enable managers and programmers to

- select a compiler best suited for their real-time application;
- identify areas where there are performance changes in a new compiler release;
- and determine the most efficient way to implement real-time algorithms.

One area that was not touched during this effort was the area of composite benchmarks. Rather than measuring individual features, this approach looks as much at the interaction between features as to the performance of the features themselves. Good examples of this approach involve the use of typical code segments from a given application collected into a program whose overall performance is measured (like the Ada Avionics Test Program Package developed by SofTech Inc.). The advantage of this approach is that for a given application domain, running this benchmark on different compiler implementations enables a more straightforward selection. The development of composite benchmarks will be addressed in a follow-on effort.

References

[1] R.M. Clapp et al., "Towards Real-time Performance Benchmarks for Ada", CACM, Vol. 29, No. 8, August 1986.

[2] N. Altman, "Factors Causing Unexpected Variations in Ada Benchmarks", Technical Report, CMU/SEI-87-TR-22, October 1987.

[3] N. Altman et al., "Timing Variation in Dual Loop Benchmarks" , Technical Report, CMU/SEI-87-TR-21, October 1987.

[4] "Catalog of Ada Runtime Implementation Dependencies", ARTEWG Report, November, 1986.

[5] A. Goel, "Evaluation of Ada Compilers Targeted to Bare Systems", To be Published,

[6] "Catalog of Interface Features and Options for the Ada Runtime Environment ", ARTEWG Report, December, 1987.

[7] M. D. Broido, "Toward Real-time Performance Benchmarks For Ada", Technical Correspondence, CACM, Vol. 30, No. 2, February 1987.

[8] L. MacLaren, "Evolving Toward Ada in Real-time Systems", Proceedings of the ACM, SIGPLAN Symposium on the Ada Programming Language, November, 1980.

[9] N. Weiderman et al., "Ada for Embedded Systems: Issues and Questions", Technical Report, CMU/SEI-87-TR-26, October 1987.

[10] SofTech Inc., "Real-time Ada", July, 1984.

[11] A. Goel, E.Wong, "Evaluation of Existing Benchmark Suites For Ada", Ada Technology Conference Proceedings, Washington DC, March 15-20,1988.

[12] Center for Software Engineering Final Report, "Establish and Evaluate Ada Runtime Features of Interest for Real-time Systems", Final Report C02092LA0003, October 1988.

[13] "Proceedings of the International Workshop on Real-time Ada Issues", UK, 13-15 May, 1987, pages 10-11.

[14] A. Tetewsky, A. Clough, R. Racine, R. Whittredge, "Mapping Ada onto Embedded Systems:Memory Constraints", Ada Letters, September/October, 1988.